Unit-1 Lecture 1

Introduction: Review of the Traditional Methodologies

A **Traditional** (software development) methodology or system development methodology in software engineering is a framework that is used to structure, plan, and control the process of developing an information system.

History

The **Traditional (**software development) methodology framework didn't emerge until the 1960s. According to Elliott (2004) the systems development life cycle (SDLC) can be considered to be the oldest formalized methodology framework for building information systems. The main idea of the SDLC has been "to pursue the development of information systems in a very deliberate, structured and methodical way, requiring each stage of the life cycle from inception of the idea to delivery of the final system, to be carried out in rigidly and sequentially". The main target of this methodology framework in the 1960s was "to develop large scale functional business systems in an age of large scale business conglomerates. Information systems activities revolved around heavy data processing and number crunching routines".

The traditional Approach

The traditional approach is also called structured methodology which is most commonly been used for so many years. Shelly, Cashman & Rosenblatt, 2006, points out that structured methodology uses a series of phases, called System development life cycle (SDLC), in order to plan, analyse, design, implement and support a computer based information system. According to Rob, 2006, System Development Life cycle (SDLC), refers to the necessary process for the successful development of the information system. These necessary processes include some phases and some activities. The phases involved in SDLC are the following.

- **System planning.**
- **System analysis.**
- **System Design.**
- **System implementation.**

• **System operation, support and security**.

Traditional approach adopts step-by-step approach to go through the above phases. The phase or activities of one step should be completed before moving to the next step. The structured approach looks the system from a top down view. George, Hoffer & Valacich, 2006, points out that the steps or phases in SDLC may vary in each organization according to its goals and it is also possible to complete some activities in one phase in parallel with some other activities of another phase. Sometimes these phases are repeated until an acceptable system is found. SDLC is traditionally pictured as a water fall model, where the end product of each phase flows sequentially to the next phase. This methodology is called waterfall model because it moves forward from one step to another in the same manner as a water fall. It is also possible to go backward from one step to another.

System planning

Dennis, Roth & Wixom, 2006, states that the system planning phase is the fundamental process of understanding why an information system should built in an organization and determining how it would be implemented. According to George, Hoffer & Valacich, 2006, first phase in the SDLC, system planning, has two primary activities.

• First someone identifies the need for a new system or notifies the desired changes in the existing system.

• Second task is to investigate the system and determine the scope of the proposed system.

In the first step In the first step, the IT department gets a formal request which is called a system request, which demands a new information system or desired changes in the existing information system. A system request can come from top managers, staffs, other departments or from the IT department itself. The second step involves a preliminary investigation to identify the nature and scope of the opportunity for a new system or the problem with the existing system. According to Shelly, Cashman & Rosenblatt, 2006, the preliminary investigation is a critical step because the outcome of this investigation will affect the entire development processes. The IT department conducts a feasibility study which is a key part of the preliminary investigation, to review anticipated costs and benefits of the proposed project. The feasibility study may recommend business process review or a IT solution depending upon the technical, operational, economic and time factors. If the preliminary investigation finds out that a full-scale system review is necessary, then the next step is system analysis.

System analysis

The purpose of the second phase of the SDLC, system analysis, is to build a logical model of the new system. According to Dennis, Roth & Wixom, 2006, system analysis phase find out information about who will use the system, what will do the system, and when and where the system will be used. The initial step in the system analysis phase is requirement modelling. The system analyst gathers the requirements of the proposed system. The requirements are gathered by using several fact finding techniques such as interviews, surveys, document reviews, observation and sampling. These fact finding results are used to build initial models. The system analysts compare these models to determine which model is most satisfying the requirements with in the cost, labour and the technical levels. So the end product of the system analysis phase is a system requirement document which describes the requirements and alternative solution for the proposed information system.

System Design

According to Shelly, Cashman & Rosenblatt, 2006, the purpose of the third phase of SDLC, system design, is to create a blue print that satisfies all the requirements documented for the proposed system. In the design phase the logical design turns into physical or technical specification. Dennis, Roth & Wixom, 2006, points out that the first step in the system design is the development of a design strategy which clarifies whether the system will be developed by the organization's own programmers, whether the development process will be outsourced to another company, or whether the company will buy an existing software package from some vendors. The application architecture is also designed during the system design phase. The application architecture will help the programmers by showing how to transform the logic design into different program modules and code. During this phase, the internal and external controls and the user interface of the system are designed. All the aspects of the system from inputs and output screens to reports, databases and computer process are designed during the system design phase. So the final product of system design is documented in the system design specification and will be submitted to review committee for approval. At the end of the design phase review committee will re-examine the feasibility analysis and project plan and will decide whether the project should continue or terminate.

System Implementation

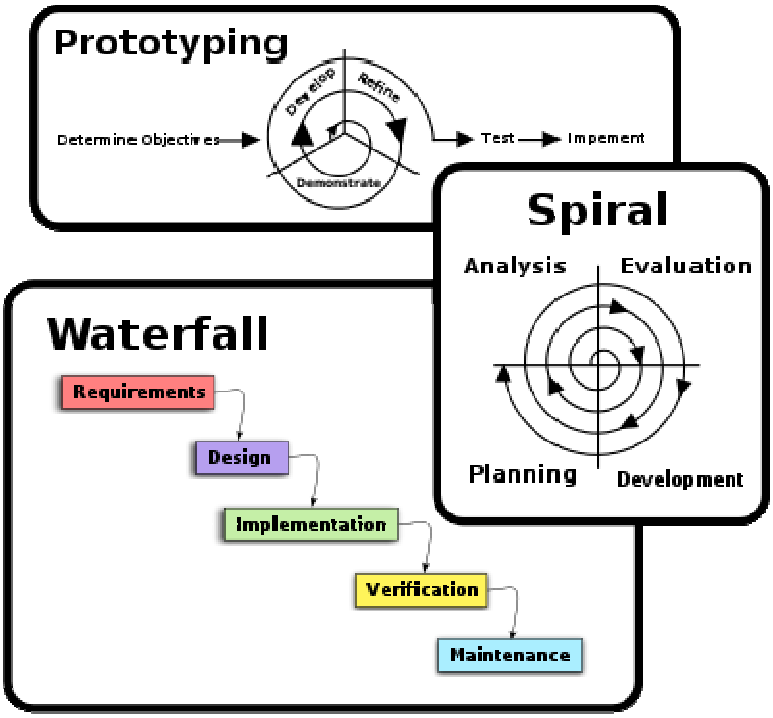
During the system implementation the system specifications turn into a working system that is tested and put into use. System implementation phase includes coding, testing and installation. If the system is purchased the system analysts configure the system and make necessary changes to meet the requirements of the organization. At the end of this phase the system is ready for use and the system becomes a part of the daily activities of the organization. Testing will be done to clarify that there is no errors associated with the system operation. Processes like converting data into system's files and training users are done at the end of this phase. A system evaluation is made at the end of this phase in order to determine whether the system operates properly and to ensure that the costs and benefits are within the expectation.

System Operation, Support and Security

The objectives of this phase are maintenance, enhancement and protection of the system. Maintenance deals with correcting errors and adapt to the changes in the business environment. Support provides options to add new features to the system to improve the efficiency. Security provide safeguard to internal and external threats. According to Shelly, Cashman & Rosenblatt, 2006, the objective of this phase is to maximize the return on the investment made for the information system.

Development Methodology

A software development methodology is a framework that is used to structure, plan, and control the process of developing an information system - this includes the pre-definition of specific deliverables and artifacts that are created and completed by a project team to develop or maintain an application.



Traditional (software development) methodology approaches

Every software development methodology framework acts as a basis for applying specific approaches to develop and maintain software. There are a number of software development approaches that have been used since the origin of information technology. These software development approaches are:

1. Waterfall Approach: linear framework type.
2. Prototyping Approach : iterative framework type
3. Incremental Approach: combination of linear and iterative framework type
4. Spiral Approach : combination of linear and iterative framework type
5. Rapid Application Development (RAD) Approach: Iterative Framework Type
6. Extreme Programming Approach.

Waterfall Approach

The Waterfall model is a sequential development approach, in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance. The first formal description of the

waterfall approach is often cited to be an article published by Winston W. Royce in 1970 although Royce did not use the term "waterfall" in this article.

Basic principles of the waterfall approach are: Project is divided into sequential phases, with some overlap and splash back acceptable between phases.

1. Emphasis is on planning, time schedules, target dates, budgets and implementation of an entire system at one time.
2. Tight control is maintained over the life of the project through the use of extensive written documentation, as well as through formal reviews and approval/signoff by the user and information technology management occurring at the end of most phases before beginning the next phase.

Prototyping Approach

Software prototyping, is the development approach of activities during software development the creation of prototypes, i.e., incomplete versions of the software program being developed.

Basic principles of the Prototyping Approach are.

1. Not a standalone, complete development methodology approach, but rather an approach to handling selected portions of a larger, more traditional development methodology (i.e. Incremental, Spiral, or Rapid Application Development (RAD)) approaches.
2. Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
3. User is involved throughout the development process, which increases the likelihood of user acceptance of the final implementation.
4. Small-scale mock-ups of the system are developed following an iterative modification process until the prototype evolves to meet the users' requirements.
5. While most prototypes are developed with the expectation that they will be discarded, it is possible in some cases to evolve from prototype to working system.
6. A basic understanding of the fundamental business problem is necessary to avoid solving the wrong problem.

Incremental Approach

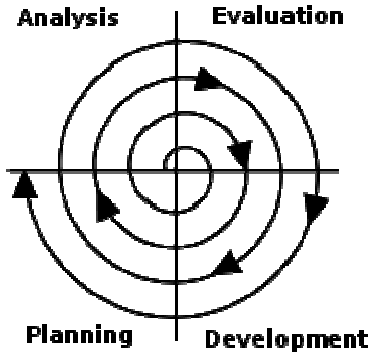
Various methods are acceptable for combining linear and iterative systems development methodology approaches, with the primary objective of each being to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.

Basic principles of the incremental development approach are:

1. A series of mini-Waterfalls are performed, where all phases of the Waterfall development approach are completed for a small part of the systems, before proceeding to the next incremental, or
2. Overall requirements are defined before proceeding to evolutionary, mini-Waterfall development approaches of individual increments of the system, or

The initial software concept, requirements analysis, and design of architecture and system core are defined using the Waterfall approach, followed by iterative Prototyping approach, which culminates in installation of the final prototype (i.e., working system).

Spiral Approach



The spiral model approach is a software development process combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts. Basic principles:

1. Focus is on risk assessment and on minimizing project risk by breaking a project into smaller segments and providing more ease-of-change during the development process, as well as providing the opportunity to evaluate risks and weigh consideration of project continuation throughout the life cycle.
2. "Each cycle involves a progression through the same sequence of steps, for each portion of the product and for each of its levels of elaboration, from an overall concept-of-operation document down to the coding of each individual program."
3. Each trip around the spiral approach traverses four basic quadrants: (1) determine objectives, alternatives, and constraints of the iteration; (2) Evaluate alternatives; Identify

and resolve risks; (3) develop and verify deliverables from the iteration; and (4) plan the next iteration.

(The figure does not reflect this and is hence wrong.)

Begin each cycle with an identification of stakeholders and their win conditions, and end each cycle with review and commitment.

Rapid Application Development (RAD) Approach

Rapid Application Development (RAD) is a software development methodology approach, which involves iterative development and the construction of prototypes. Rapid application development is a term originally used to describe a software development process introduced by James Martin in 1991.

Basic principles:

1. Key objective is for fast development and delivery of a high quality system at a relatively low investment cost.
2. Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
3. Aims to produce high quality systems quickly, primarily through the use of iterative Prototyping (at any stage of development), active user involvement, and computerized development tools. These tools may include Graphical User Interface (GUI) builders, Computer Aided Software Engineering (CASE) tools, Database Management Systems (DBMS), fourth-generation programming languages, code generators, and object-oriented techniques.
4. Key emphasis is on fulfilling the business need, while technological or engineering excellence is of lesser importance.
5. Project control involves prioritizing development and defining delivery deadlines or "time boxes". If the project starts to slip, emphasis is on reducing requirements to fit the time box, not in increasing the deadline.
6. Generally includes Joint Application Development (JAD), where users are intensely involved in system design, either through consensus building in structured workshops, or through electronically facilitated interaction.
7. Active user involvement is imperative.
8. Iteratively produces production software, as opposed to a throwaway prototype.
9. Produces documentation necessary to facilitate future development and maintenance.
10. Standard systems analysis and design techniques can be fitted into this framework.

Some Traditional Methods

1. Information Engineering (IE)
2. Information systems activity and change analysis (ISAC)
3. Structured Analysis/Structured Design (SASD)
4. Structured Systems Analysis and Design Methods (SSADM)
5. Structured Analysis and Design Technique (SADT)
6. Jackson System Development (JSD)
7. Nijssen's Information Analysis Method (NIAM)
8. Mascot-3

Unit-1 Lecture 2

Advantages of Object Oriented Methodologies over Traditional Methodologies

1. Object Oriented Methodology closely represents the problem domain. Because of this, it is easier to produce and understand designs.
2. The objects in the system are immune to requirement changes. Therefore, allows changes more easily.
3. The objects in the system are immune to requirement changes. Therefore, allows changes more easily.
4. Object Oriented Methodology designs encourage more re-use. New applications can use the existing modules, thereby reduces the development cost and cycle time.
5. Object Oriented Methodology approach is more natural. It provides nice structures for thinking and abstracting and leads to modular design.

Comparison between traditional approach and object oriented approach

Both traditional approach and object oriented approach have its own strengths and weaknesses. The primary difference between both these approaches is how a problem is decomposed. Traditional methodology is either data centric or process centric. But object oriented views system as collection of both data and processes. Object oriented approach use UML to describe the system concept as a collection of objects which incorporate both data and process. Traditional methodology introduces the use of modelling techniques to describe system's basic business processes and the data that support the processes. The traditional methodology uses one set of diagram to represent the process which is called process model and a separate set of diagram to represent data which is called data model. Since both process model and data model are important system analyst must decide which set to develop first and use as the core of the system. One of the advantages with waterfall model is that the system requirements are identified long before the beginning of the programming phase. The changes to the requirements become minimized as the system development proceeds. But the problem with the structured approach is the design of the proposed system should be completely specified prior to the programming phase of the system development. This causes a long time gap between completion of the system proposal and delivery of the final system which results in

poor communication mechanism. Because of this time gap between the initial proposal of the system and the final introduction, the users will not be prepared for the new system. In object oriented system a functional model of real world system is developed in the analysis phase itself. This allows the users to get information about what the final system must do. The use messages provide better communication between the objects. The encapsulation property of the object oriented methodology allows reuse of the code in different system. This results in significant cost reduction and faster development time. Another characteristic that differentiate both the methodology is the use of different models and designs to describe the system. Structured methodology mainly uses three models to describe the system which are Data Flow Diagram (DFD), Entity Relationship Diagram (ERD) and structured chart. Object oriented methodology uses about thirteen diagrams and models like class diagram, object diagram, sequence diagram, component diagram, deployment diagram etc. The complexity of diagram used in object oriented approach is much higher than the diagram used in traditional approach. During the implementation phase of traditional approach, the structured chart is used to show the interaction between program modules. In contrast, with object oriented approach, the relationship between the objects is already defined during the analysis phase itself by means of different models and diagrams. In traditional approach there are clear cut steps to move from one phase to another and at the end of each phase some documentation will be produced regarding to the activities done in that particular phase. But in object oriented approach not only there is no clear cut step in moving from analysis to design and from design to implementation, but also the steps are iterative. This is time consuming and results in repetitive testing throughout the development process of the system. This continues analysis of the system during the development phase offers a much better end product, a system with better efficiency and minimum errors. In object oriented approach there will not be any documents per phase and all the information about the activities in that particular phase will be represented within the model description. In traditional methodology, system components such as inputs, outputs, database, files and program modules are designed from the data flow diagram (DFD). But in object oriented methodology the information about the input and output are scattered in different diagrams and models. While in the traditional approach, data structures associated with DFD are used to describe data, object oriented approach uses separate classes to describe the data. The following table summarises these different characteristics and also points out the strengths and weakness of both the approach.

Traditional Approach	Object oriented approach
1) Either data-centric or process- centric	1) Views system as a collection of self contained objects, including both data and process
2) Views a system as a top-down approach to system development	2) Views a system as a bottom-up approach to system development
3) Adopts step by step approach to SDLC phases	3) No clear cut step form one phase to another
4) Specific documentations are produced at the end of each phase	4) There is no specific documentation after each phase.
5) Mainly three models or diagram are used	5) Uses about thirteen models/diagrams
6) Data needed for designing and development of the system is in a single data dictionary	6) No single repository for data
7) Input and output of the system are identified from the DFD	7) Input and output are scattered in many sequence diagram

Unit-1 Lecture 3

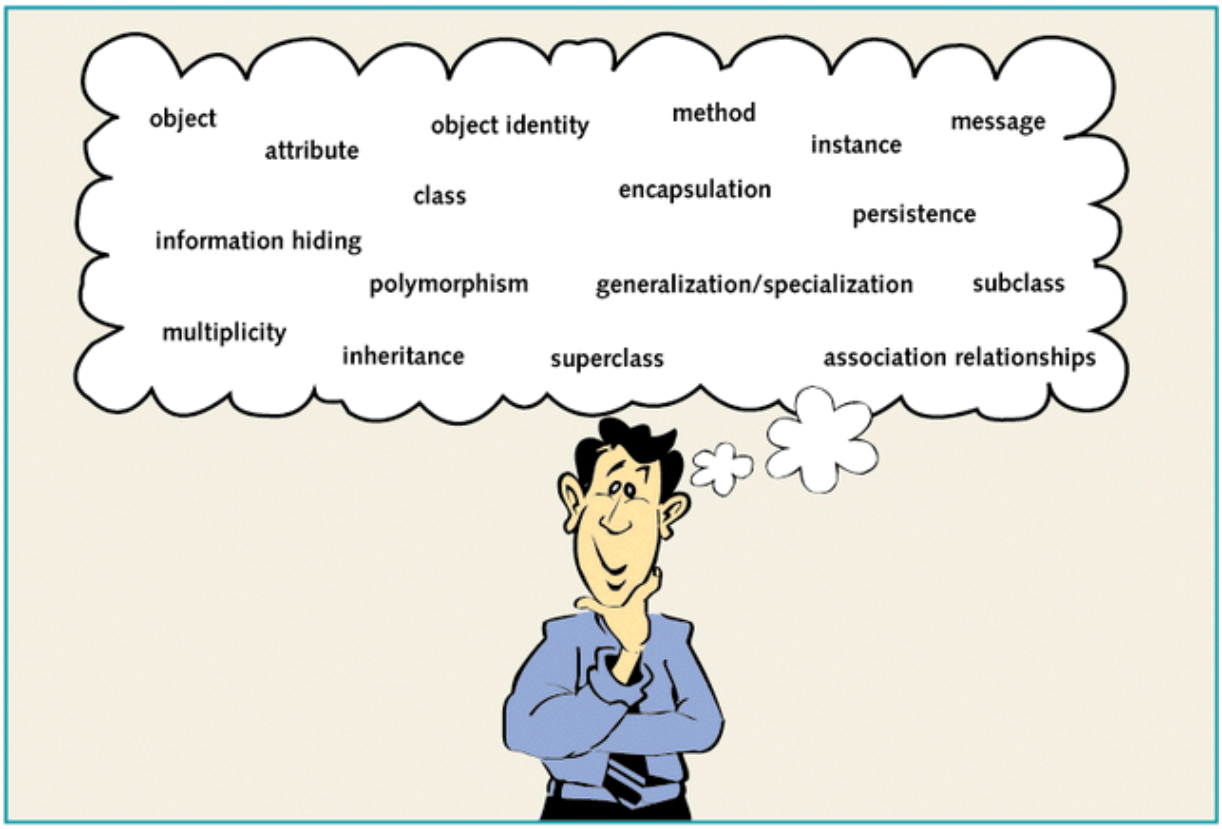


Figure 1-5 Key OO concepts

Classes, Objects, Encapsulation, Association

Object-Oriented Concept

To define the CIM models and manage them well, understand the concept of Object Orientation is an important work since the MOF is based on the object-oriented concept. In this lecture we will introduce the concept of object-orientation, class modeling with object-oriented, and the critical concept in object-oriented programming and finally, we will introduce how to use Rational Rose to design the model.

Basic Concept

Object-Orientation is so called because this method sees things that are part of the real world as objects. A phone is an object in the same way as a bicycle, a human being, or an insurance policy is objects. In everyday life, we simplify objects in our thinking – we work with models. Software development does essentially the same: objects occurring in reality are reduced to a few features that are relevant in the current situation. Instead of real objects, we work with symbols. Properties and composition of objects correspond only roughly to reality; the model selects only those aspects that are useful for carrying out a specific task. In figure 3.1, we use model to describe a reality that a teacher owns a bicycle and he reads a book.

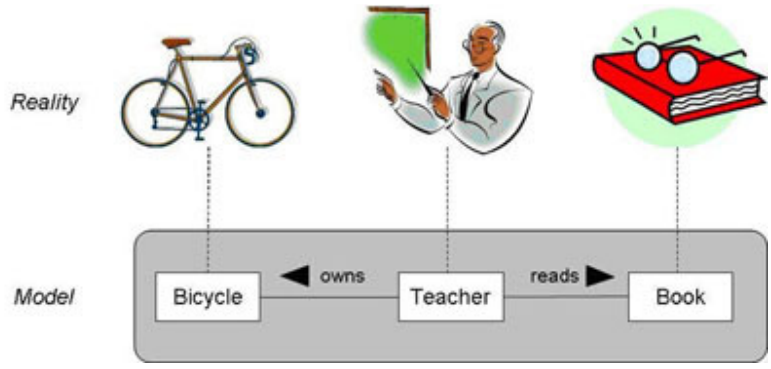


Figure 3.1 Complex realities are made more manageable by abstract models.

Data Abstraction

Abstraction is the presentation of simple concept (or object) to the external world. Abstraction concept is to describe a set of similar concrete entities. It focuses on the essential, inherent aspects of an entity and ignoring its accidental properties. The abstraction concept is usually used during analysis: deciding only with application-domain concepts, not making design and implementation decisions.

Two popular abstractions: **procedure abstraction** and **data abstraction**. Procedure abstraction is to decompose problem to many simple sub-works. Data abstraction is to collect essential elements composing to a compound data. These two abstractions have the same objective: reusable and replaceable. Figure 3.2 shows an example of data abstraction to abstract doors as a data structure with essential properties.

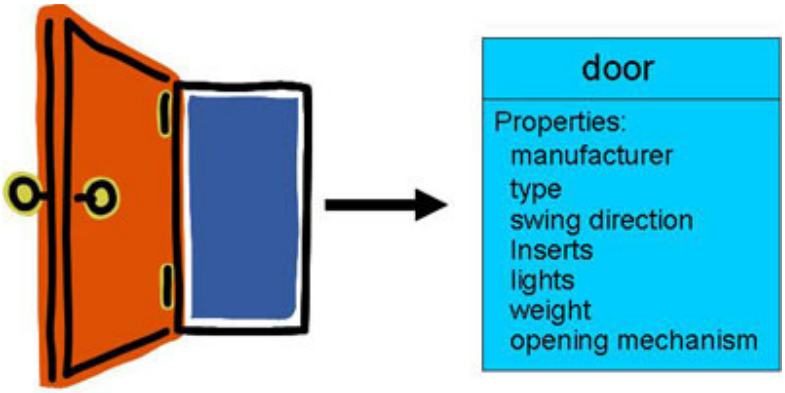


Figure 3.2 Example of Data Abstraction.

Encapsulation

Encapsulation means as much as shielding. Each object-oriented object has a shield around it. Objects can't 'see' each other. They can exchange things though, as if they are interconnected through a hatch. Figure 3.3 shows the concept of the encapsulation. It separates the external aspects of an object from the internal implementation details of the

object, which are hidden from other objects. The object encapsulates both data and the logical procedures required to manipulate the data.

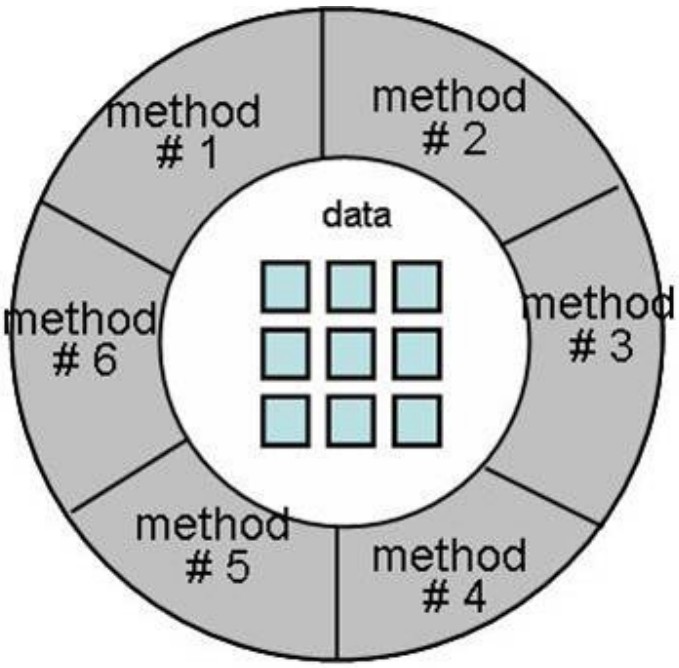


Figure 3.3 Example of Encapsulation Concept.

Class Modeling

Class and Object Instance

A class is used to describe something in the world, such as occurrences, things, external entities, roles, organization units, places or structures. A class describes the structure and behavior of a set of similar objects. It is often described as a template, generalized description, pattern or blueprint for an object, as opposed to the actual object, itself. Once a class of items is defined, a specific instance of the class can be defined. An instance is also called "object." Figure 3.4 shows an example of teacher class and its object in UML notation.

The following table gives some examples of classes and objects

Type	Example of class	Example of object
Occurrence	Alarm	Fire alarm
Things	Car	Ferrari 360
External entities	Door	Fire door
Roles	Teacher	John, Nick
Organizational units	IECS department	FCU_IECS,

Figure 3.4 Class and Object.

A class is a template for an object, a user-defined datatype that contains variables,properties of an object. A class defines abstract characteristics of a thing (object), including its characteristics (its **attributes**, **fields** or **properties**) and the **things it can do** (behaviors, **methods**, **operations** or **features.** For example, the class Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors). Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable to a non-programmer familiar with the problem domain, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using **encapsulation**). Collectively, the properties and methods defined by a class are called **members**.

Property and Method

Properties in a class are used to present the structure of the objects: their components and the information or data contained therein. (e.g., name, owner, ground clearance).An instance of a class has the properties defined in its class and all of the classes from which its class inherits. In figure 3.5, name, weight and breed are three properties of the class Dog.

Methods in a class describe the behavior of the objects. It represents a function that an instance of the class can be asked to perform. In the figure 3.5, a Dog class defines three methods: sit, beg and run which indicate the behavior a dog can have.

Figure 3.5 An example of a Dog Class

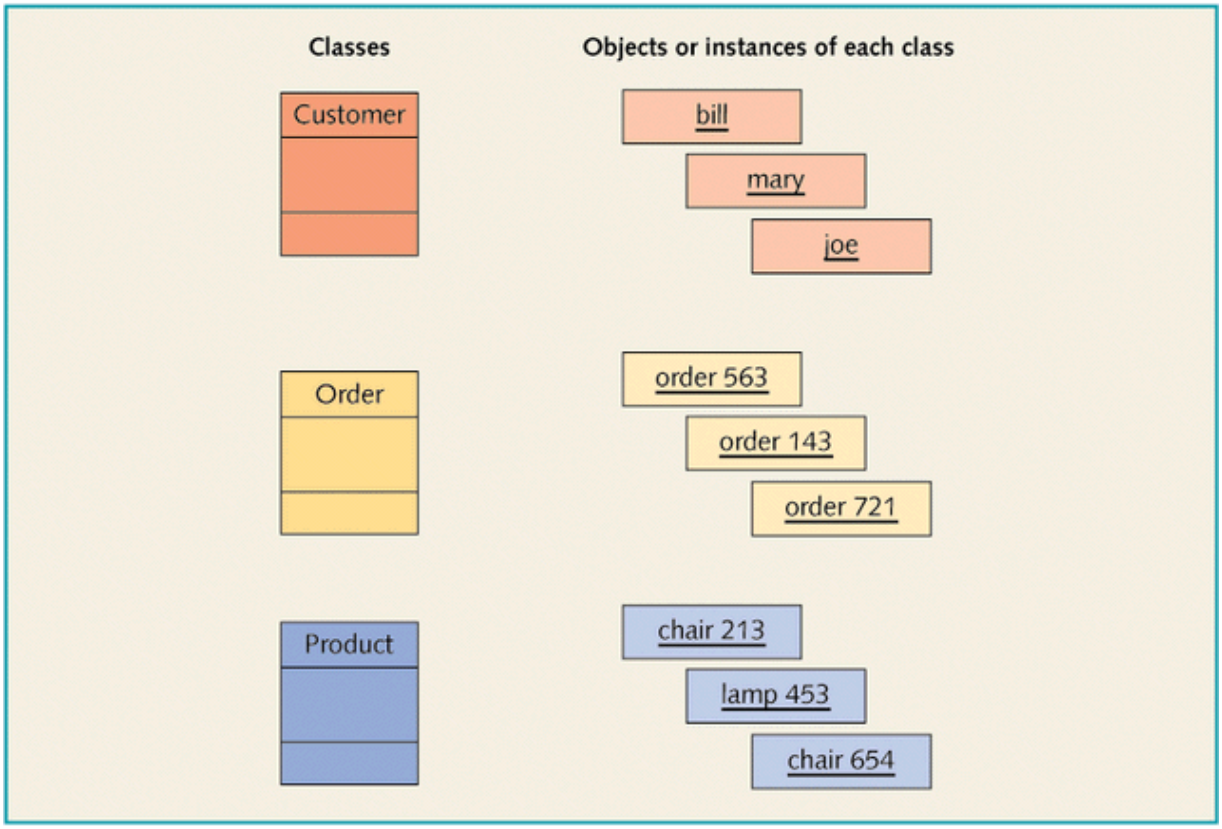


Figure 1-9 Class versus objects or instances of the class

Object Instance

1. **Object attributes have specific values and boundaries.**
2. **Object Class describes a group of objects with similar properties.**
3. **Object Diagram provides a formal graphic notation for modeling object, classes and their relationships to one another.**

Unit-1 Lecture 5

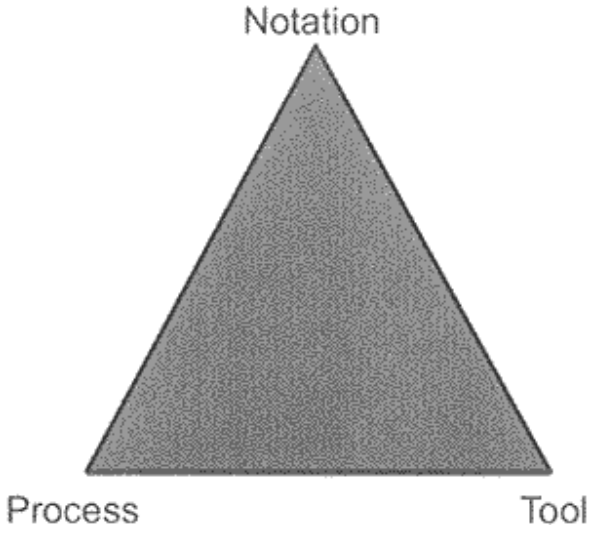
What is Visual Modeling?

Visual Modeling is a way of thinking about problems using models organized around real-world ideas. Models are useful for understanding problems, communicating with everyone involved with the project (customers, domain experts, analysts, designers, etc.), modeling enterprises, preparing documentation, and designing programs and databases. Modeling promotes better understanding of requirements, cleaner designs, and more maintainable systems. Models are abstractions that portray the essentials of a complex problem or structure by filtering out nonessential details, thus making the problem easier to understand. Abstraction is a fundamental human capability that permits us to deal with complexity. Engineers, artists, and craftsmen have built models for thousands of years to try out designs before executing them. Development of software systems should be no exception. To build complex systems; the developer must abstract different views of the system, build models using precise notations, verify that the models satisfy the requirements of the system, and gradually add detail to transform the models into an implementation. We build models of complex systems because we cannot comprehend such systems in their entirety. There are limits to the human capacity to understand complexity. This concept may be seen in the world of architecture. If you want to build a shed in your backyard, you can just start building; if you want to build a new house, you probably need a blueprint; if you are building a skyscraper, you definitely need a blueprint. The same is true in the world of software. Staring at lines of source code or even analyzing forms in Visual Basic do little to provide the programmer with a global view of a development project. Constructing a model allows the designer to focus on the big picture of how a project's components interact, without having to get bogged down in the specific details of each component. Increasing complexity, resulting from a highly competitive and ever-changing business environment, offers unique challenges to system developers. Models help us organize, visualize, understand, and create complex things. They are used to help us meet the challenges of developing software today and in the future.

The Triangle for Success

I have often used the triangle for success as shown in Figure 1-1 to explain the components needed for a successful project. You need all three facets—a notation, a process, and a tool. You can learn a notation, but if you don't know how to use it (process), you will probably fail. You may have a great process, but if you can't communicate the process (notation), you will probably fail. And lastly, if you cannot document the artifacts of your work (tool), you will probably fail.

Figure 1-1. Triangle for Success



The Role of Notation

Notation plays an important part in any model—it is the glue that holds the process together. Notation has three roles:

• It serves as the language for communicating decisions that are not obvious or cannot be inferred from the code itself.

• It provides semantics that are rich enough to capture all important strategic and tactical decisions.

• It offers a form concrete enough for humans to reason and for tools to manipulate.

The Unified Modeling Language (UML) provides a very robust notation, which grows from analysis into design. Certain elements of the notation (for example, classes, associations, aggregations, inheritance) are introduced during analysis. Other elements of the notation (for example, containment implementation indicators and properties) are introduced during design.

Unit-1 Lecture 6

Introduction to Unified Modeling Language (UML)

History of the UML

During the 1990s many different methodologies, along with their own set of notations, were introduced to the market. Three of the most popular methods were OMT (Rumbaugh), Booch, and OOSE (Jacobson). Each method had its own value and emphasis. OMT was strong in analysis and weaker in the design area. Booch 1991 was strong in design and weaker in analysis. Jacobson was strong in behavior analysis and weaker in the other areas. As time moved on, Booch wrote his second book, which adopted a lot of the good analysis techniques advocated by Rumbaugh and Jacobson, among others. Rumbaugh published a series of articles that have become known as OMT-2 that adopted a lot of the good design techniques of Booch. The methods were beginning to converge but they still had their own unique notations. The use of different notations brought confusion to the market since one symbol meant different things to different people. For example, a filled circle was a multiplicity indicator in OMT and an aggregation symbol in Booch. You will hear the term "method wars" being used to describe this period of time—is a class a cloud or a rectangle? Which one is better? The end of the method wars as far as notation is concerned comes with the adoption of the Unified Modeling Language (UML). "UML is a language used to specify, visualize, and document the artifacts of an object-oriented system under development. It represents the unification of the Booch, OMT, and Objectory notations, as well as the best ideas from a number of other methodologists as shown in Figure 1-2. By unifying the notations used by these object-oriented methods, the Unified Modeling Language provides the basis for a de facto standard in the domain of object-oriented analysis and design founded on a wide base of user experience."

The UML is an attempt to standardize the artifacts of analysis and design: semantic models, syntactic notation, and diagrams. The first public draft (version 0.8) was introduced in October

1995. Feedback from the public and Ivar Jacobson's input were included in the next two versions (0.9 in July 1996 and 0.91 in October 1996). Version 1.0 was presented to the Object Management Group (OMG) for standardization in July 1997. Additional enhancements were incorporated into the 1.1 version of UML, which was presented to the OMG in September 1997. In November 1997, the UML was adopted as the standard modeling language by the OMG.

Overview

The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

1. activities
2. actors
3. business processes
4. database schemas
5. (logical) components
6. programming language statements
7. reusable software components.

UML combines techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies. UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language. UML aims to be a standard modeling language which can model concurrent and distributed systems. UML is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG).

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages. UML is extensible, with two mechanisms for customization: profiles and stereotypes.

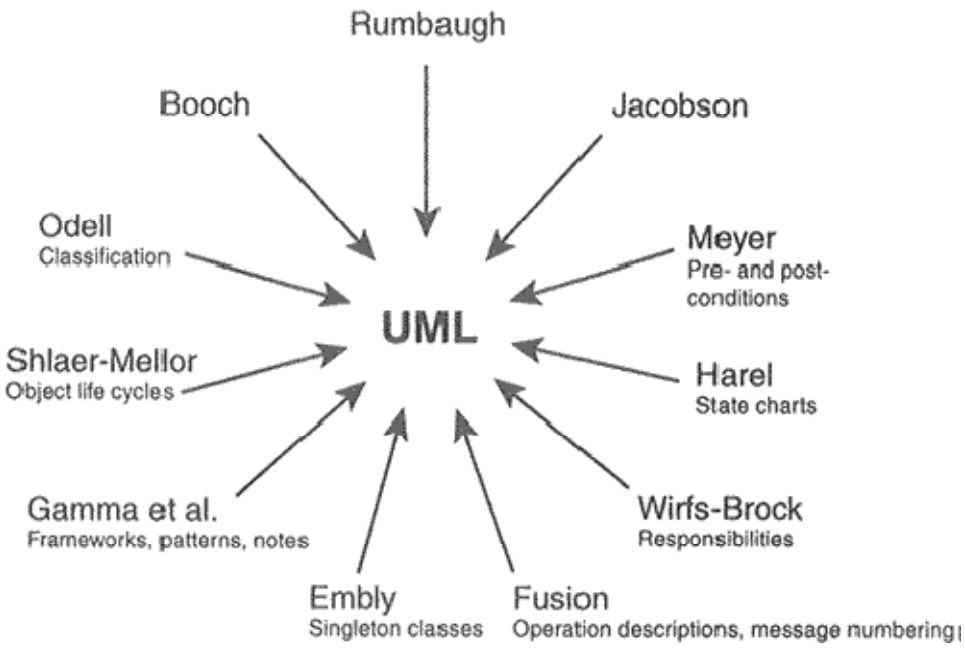


Figure 1-2. UML Inputs

Unit-1 Lecture 7

Introduction to Rational Rose CASE tool

Any software development method is best supported by a tool. There are many tools on the market today— everything from simple drawing tools to sophisticated object modeling tools. At every step, there is a description of how to use Rational Rose to complete the step. The Rational Rose product family is designed to provide the software developer with a complete set of visual modeling tools for development of robust, efficient solutions to real business needs in the client/server, distributed enterprise, and real-time systems environments. Rational Rose products share a common universal standard, making modeling accessible to nonprogrammers wanting to model business processes as well as to programmers modeling applications logic. An evaluation

version of the Rational Rose tool may be obtained at the Rational Software Corporation website at www.rational.com. Rational Rose tool, a subset of the diagrams is available in the Microsoft Visual Modeler tool. Microsoft Visual Modeler is the entry-level product, designed specifically for the first-time modeler. The tool provides the capability to

• Identify and design business objects, and then map them to software components

• Partition services across a three-tiered service model

• Design how components will be distributed across a network

• Generate Visual Basic code frameworks directly from your model

• Use reverse engineering to create models from existing components and applications

• Use round-trip engineering facilities to keep your designs synchronized with your code

Rational Rose extends Visual Modeler for dynamic behaviors, such as business requirements analysis, business scenario analysis with sequence and collaboration diagrams, state modeling, and additional code generation capabilities for DDL and IDL, along with the inclusion of a scripting language to provide access to the Rose domain.

Rational Rose is an object-oriented Unified Modeling Language (UML) software design tool intended for visual modeling and component construction of enterprise-level software applications. In much the same way a theatrical director blocks out a play, a software designer uses Rational Rose to visually create (model) the framework for an application by blocking out classes with actors (stick figures), use case elements (ovals), objects (rectangles) and messages/relationships (arrows) in a sequence diagram using drag-and-drop symbols. Rational Rose documents the diagram as it is being constructed and then generates code in the designer's choice of C++, Visual Basic, Java, Oracle8, CORBA or Data Definition Language.

Two popular features of Rational Rose are its ability to provide iterative development and round-trip engineering. Rational Rose allows designers to take advantage of iterative development (sometimes called evolutionary development) because the new application can be created in stages with the output of one iteration becoming the input to the next. (This is in contrast to waterfall development where the whole project is completed from start to finish before a user gets to try it out.) Then, as the developer begins to understand how the components interact and makes modifications in the design, Rational Rose can perform what is called "round-trip engineering" by going back and updating the rest of the model to ensure the code remains consistent.

Rational Rose is extensible, with downloadable add-ins and third-party partner applications. It supports COM/DCOM (ActiveX), JavaBeans, and Corba component standards.

Rational Rose is commercial case-tool software. It supports two essential elements of modern software engineering: component based development and controlled iterative development. Models created with Rose can be visualized with several UML diagrams. Rose also supports Round-Trip engineering with several languages.

How was the tool used

The tool was used to create the packet hierarchy, structure and dependencies of classes', state diagram and an example of the control flow (sequence diagram) of the coffee maker software. C++-code was generated from the created model using the Rose's C++ code generator. This created code was then altered (due to changes to requirements) and reverse engineered to a new model or to an update of the existing model.

Rational Rose- Positive factors

1. The tool itself was quite easy to install.
2. The creation of the different diagrams can be learned quite fast.
3. Code generation is simple.
4. C++ Analyzer was also easy to use (though it's functionality could be included in the Rose itself)

Rational Rose- Negative factors

1. At first the tool seems to be quite complex.
2. Some minor bugs were found.
3. Separate tool had to be used (and learned) to reverse-engineer files.
4. Layout manager could have been a bit more effective.
5. Generated code was a bit obfuscated.

Unit-2 Lecture 9

The Role of Process

A successful development project satisfies or exceeds the customer's expectations, is developed in a timely and economical fashion, and is resilient to change and adaptation. The development life cycle must promote creativity and innovation. At the same time, the development process must be controlled and measured to ensure that the project is indeed completed. "Creativity is essential to the crafting of all well-structured object-oriented architectures, but developers allowed completely unrestrained creativity tend to never reach closure. Similarly, discipline is required when organizing the efforts of a team of developers, but too much discipline gives birth to an ugly bureaucracy that kills all attempts at innovation." A well-managed iterative and incremental life cycle provides the necessary control without affecting creativity.

What is Iterative and Incremental Development

In an iterative and incremental life cycle (Figure 1-3), development proceeds as a series of iterations that evolve into the final system. Each iteration consists of one or more of the following process components: business modeling, requirements, analysis, design, implementation, test, and deployment. The developers do not assume that all requirements are known at the beginning of the life cycle; indeed change is anticipated throughout all phases.

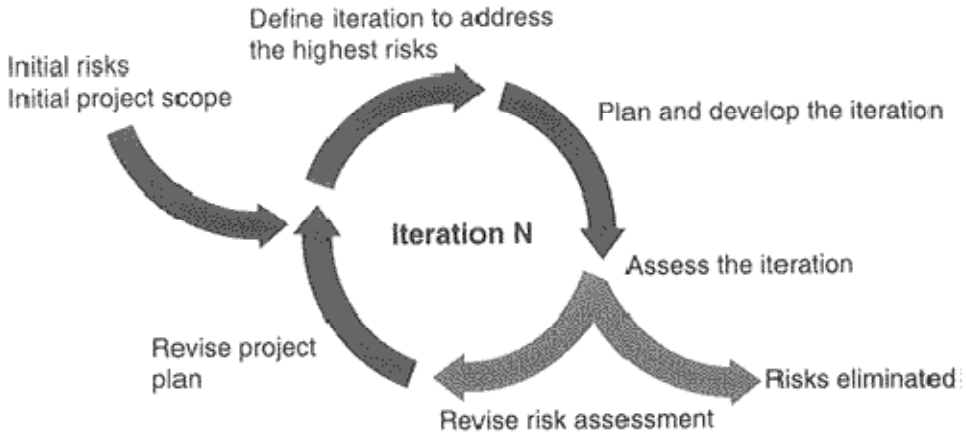


Figure 1-3. Iterative and Incremental Development

This type of life cycle is a risk-mitigating process. Technical risks are assessed and prioritized early in the life cycle and are revised during the development of each iteration. Risks are attached to each iteration so that successful completion of the iteration alleviates the risks attached to it. The releases are scheduled to ensure that the highest risks are tackled first. Building the system in this fashion exposes and mitigates the risks of the system early in the life cycle. The result of this life cycle approach is less risk coupled with minimal investment.?

The Rational Unified Process

Control for an iterative and incremental life cycle is supported by employing the Rational Unified Process—an extensive set of guidelines that address the technical and organizational aspects of software development focusing on requirements analysis and design.
The Rational Unified Process is structured along two dimensions:
• Time—division of the life cycle into phases and iterations
• Process components—production of a specific set of artifacts with well-defined activities

Both dimensions must be taken into account for a project to succeed. Structuring a project along the time dimension involves the adoption of the following time-based phases:

• Inception—specifying the project vision

• Elaboration—planning the necessary activities and required resources; specifying the features and designing the architecture.

• Construction—building the product as a series of incremental iterations

• Transition—supplying the product to the user community (manufacturing, delivering, and training)

Structuring the project along the process component dimension includes the following activities:
• Business modeling—the identification of desired system capabilities and user needs.

• Requirements—a narration of the system vision along with a set of functional and nonfunctional requirements.

• Analysis and Design—a description of how the system will be realized in the implementation phase.

• Implementation—the production of the code that will result in an executable system.

• Test—the verification of the entire system.

• Deployment—the delivery of the system and user training to the customer.

Figure 1-4 shows how the process components are applied to each time-based phase.

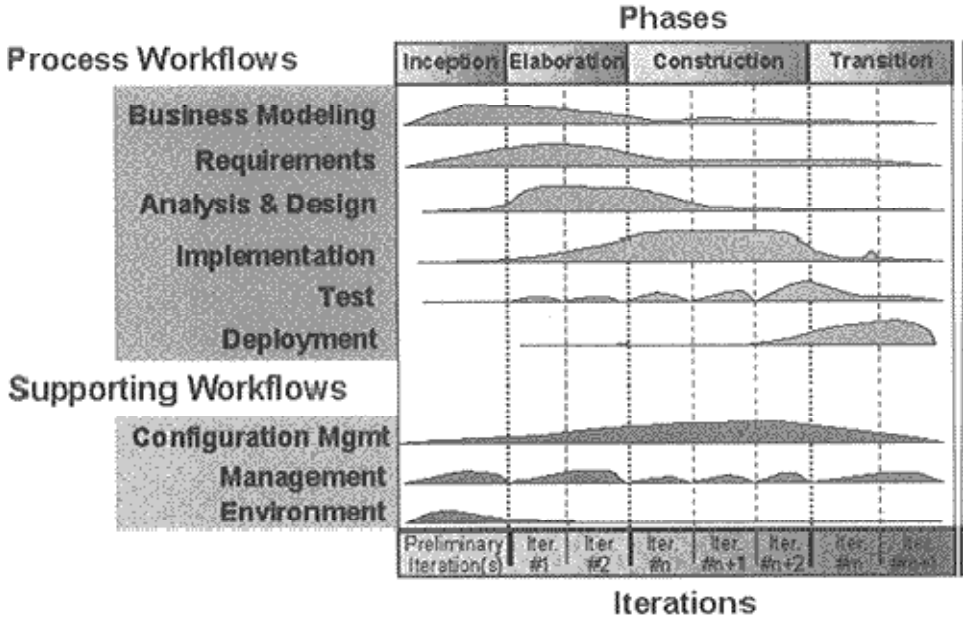


Figure 1-4. The Development Process

Each activity of the process component dimension typically is applied to each phase of the time-based dimension. However, the degree to which a particular process component is applied is dependent upon the phase of development. For example, you may decide to do a proof of concept prototype during the Inception Phase, and thus, you will be doing more than just capturing requirements (you will be doing the analysis, design, implementation, and test needed to complete the prototype). The majority of the analysis process component occurs during the Elaboration Phase. However, it is also advisable to complete the first few iterations of the system during this phase. These first few iterations typically are used to validate the analysis decisions made for the architecture of the system. Hence, you are doing more than just analyzing the problem. During the

Construction Phase of development, the system is completed as a series of iterations. As with any type of development structure, things always crop up as the system is built; thus, you are still doing some analysis.

The diagram is meant to be a guideline for the life cycle of your project. The main point is if you are still trying to figure out what you are supposed to be building as you are writing the code, you are probably in trouble. You should also note that testing is applied throughout the iteration process—you do not wait until all the code is done to see if it all works together!

Unit-2 Lecture 11

Creating Use Case Diagrams

Actors

Actors are not part of the system—they represent anyone or anything that must interact with the system. An actor may:-

• Only input information to the system

• Only receive information from the system

• Input and receive information to and from the system

Typically, these actors are found in the problem statement and by conversations with customers and domain experts. The following questions may be used to help identify the actors for a system:

• Who is interested in a certain requirement?

• Where in the organization is the system used?

• Who will benefit from the use of the system?

• Who will supply the system with this information, use this information, and remove this information?

• Who will support and maintain the system?
• Does the system use an external resource?

• Does one person play several different roles?

• Do several people play the same role?

• Does the system interact with a legacy system?

.

In the UML, an actor is represented as a stickman, as shown in Figure 3-1

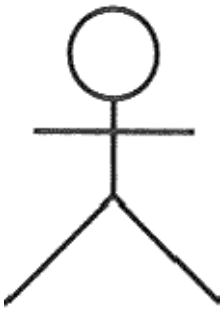


Figure 3-1. UML Notation for an Actor

What Constitutes a "Good" Actor?

Care must be taken when identifying the actors for a system. This identification is done in an iterative fashion—the first cut at the list of actors for a system is rarely the final list. For example, is a new student a different actor than a returning student? Suppose you initially say the answer to this question is yes. The next step is to identify how the actor interacts with the system. If the new student uses the system differently than the returning student, they are different actors. If they use the system in the same way, they are the same actor. Another example is the creation of an actor for every role a person may play. This may also be overkill. A good example is a teaching assistant in the ESU Course Registration System. The teaching assistant takes classes and teaches classes. The capabilities needed to select courses to take and to teach are already captured by the identification of functionality needed by the Student and the Professor actors. Therefore, there is no need for a Teaching Assistant actor. By looking at the identified actors and documenting how they use the system, you will iteratively arrive at a good set of actors for the system.

Actors in the ESU Course Registration System

The previous questions were answered as follows:

• Students want to register for courses

• Professors want to select courses to teach

• The Registrar must create the curriculum and generate a catalog for the semester

• The Registrar must maintain all the information about courses, professors, and students

• The Billing System must receive billing information from the system

Creating Actors in Rational Rose

1. Right-click on the Use Case View package in the browser to make the shortcut menu visible.

2. Select the New: Actor menu option. A new actor called New Class is placed in the browser.

3. With the actor called New Class selected, enter the desired name of the actor.

The Browser view of the actors for the ESU Course Registration System is shown in Figure 3-2.

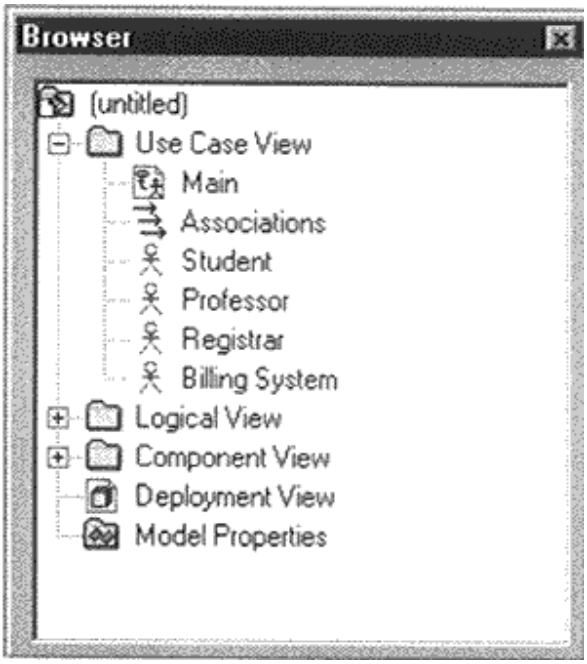


Figure 3-2. Actors

Actor Documentation

A brief description for each actor should be added to the model. The description should identify the role the actor plays while interacting with the system.

The actor descriptions for the ESU Course Registration System are:

• **Student—**a person who is registered to take classes at the University

• **Professor—**a person who is certified to teach classes at the University

• **Registrar—**the person who is responsible for the maintenance of the ESU Course Registration System

• **Billing System—**the external system responsible for student billing

Documenting Actors in Rational Rose

1. If the documentation window is not visible, open the documentation window by selecting the Documentation menu choice from the View menu.

2. Click to select the actor in the browser.

3. Position the cursor in the documentation window and enter the documentation.

The documentation for the Student actor is shown in Figure 3-3.

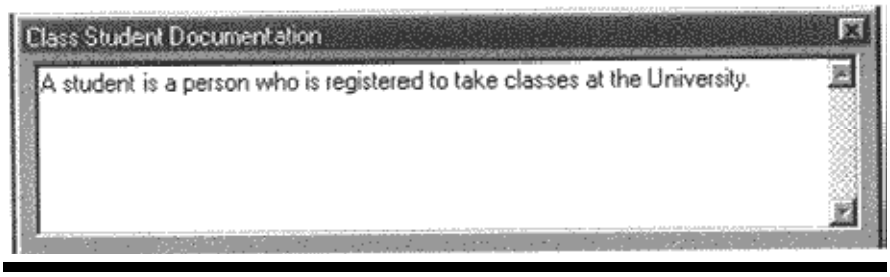


Figure 3-3. Student Actor Documentation

Use Cases

Use cases model a dialogue between an actor and the system. They represent the functionality provided by the system; that is, what capabilities will be provided to an actor by the system. The collection of use cases for a system constitutes all the defined ways the system may be used. The formal definition for a use case is: A use case is a sequence of transactions performed by a system that yields a measurable result of values for a particular actor.
The following questions may be used to help identify the use cases for a system:

• What are the tasks of each actor?

• Will any actor create, store, change, remove, or read information in the system?

• What use case will create, store, change, remove, or read this information?

• Will any actor need to inform the system about sudden, external changes?

• Does any actor need to be informed about certain occurrences in the system?

• What use cases will support and maintain the system?

• Can all functional requirements be performed by the use cases?

In the UML, a use case is represented as an oval, as shown in Figure 3-4.

Figure 3-4. UML Notation for a Use Case

What Constitutes a "Good" Use Case?

Over the years there has been a lot of discussion dealing with the "goodness" of a use case. One problem that I have encountered is the level of detail found in use cases. That is, how big (or how little) should they be? There is no one, right answer. The rule of thumb that I apply is the following: A use case typically represents a major piece of functionality that is complete from beginning to end. A use case must deliver something of
value to an actor.
For example, in the **ESU Course Registration System**, the student must select the courses for a semester, the student must be added to the course offerings, and the student must be billed. Is this

three use cases, or just one? I would make it one because the functionality represents what happens from beginning to end. What good would the system be if a student was not added to the courses selected (or at least notified if the addition does not occur)? Or if the student was not billed (the University would not stay in business if all courses were free!)? Another problem is how to bundle functionality that is different but seems to belong together. For example, the Registrar must add courses, delete courses, and modify courses. Three use cases or one use case? Here again, I would make one use case—the maintenance of the curriculum, since the functionality is started by the same actor (the Registrar) and deals with the same entities in the system (the curriculum).

Use Cases in the ESU Course Registration System

The following needs must be addressed by the system:

• The Student actor needs to use the system to register for courses.

• After the course selection process is completed, the Billing System must be supplied with billing information.

• The Professor actor needs to use the system to select the courses to teach for a semester, and must be able to receive a course roster from the system.

• The Registrar is responsible for the generation of the course catalog for a semester, and for the maintenance of all information about the curriculum, the students, and the professors needed by the system.

Based on these needs, the following use cases have been identified:

• Register for courses

• Select courses to teach

• Request course roster

• Maintain course information

• Maintain professor information

• Maintain student information

• Create course catalog

Creating Use Case in Rational Rose

1. Right-click on the Use Case View in the browser to make the shortcut menu visible.
2. Select the New: Use Case menu option. A new unnamed use case is placed in the browser.
3. With the use case selected, enter the desired name of the use case.

The browser view of the use cases for the ESU Course Registration System is shown in Figure 3-5

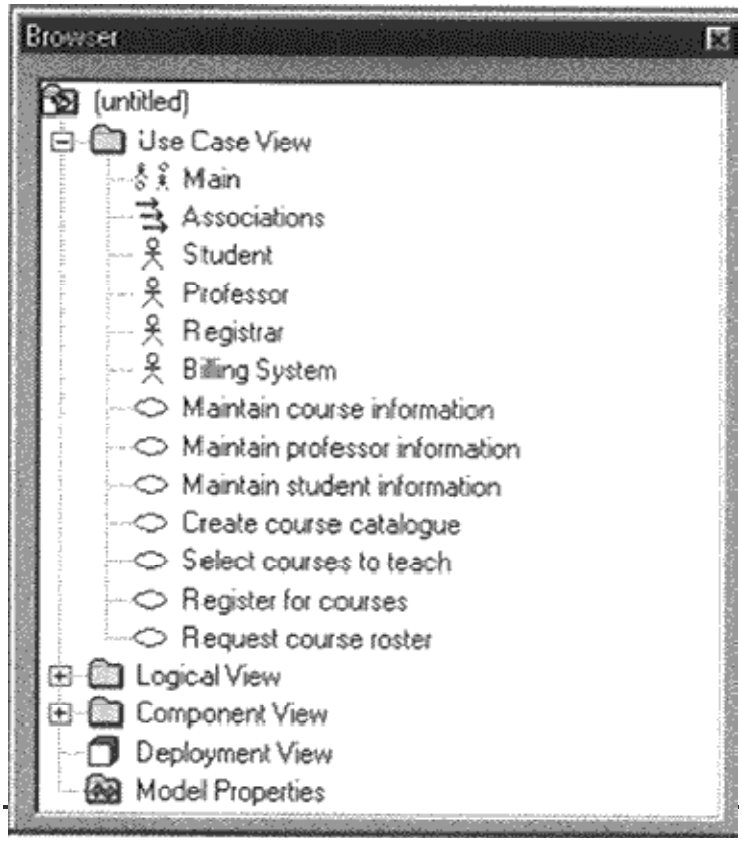


Figure 3-5. Use Cases

Brief Description of a Use Case

The brief description of a use case states the purpose of the use case in a few sentences, providing a high-level definition of the functionality provided by the use case. This description typically is created during the Inception Phase as the use case is identified.

The brief description of the *Register for Courses* use case is as follows:

This use case is started by the Student. It provides the capability to create, modify, and/or review a student schedule for a specified semester.

Creating a Use Case Brief Description in Rational Rose

1. Click to select the use case in the browser.

2. Position the cursor in the documentation window and enter the brief description for the use case. If the documentation window is not visible, select the View: Documentation menu choice to make the window visible.

The brief description of the *Register for Courses* use case is shown in Figure 3-6.

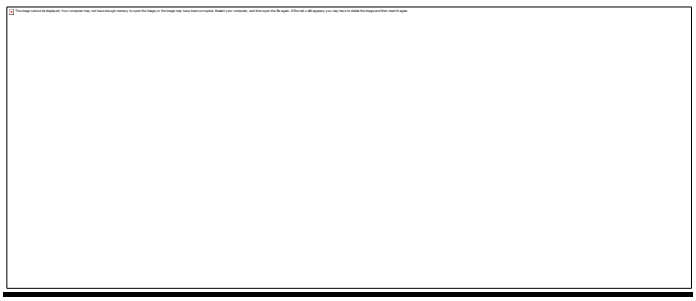


Figure 3-6. Use Case Brief Description

The Flow of Events for a Use Case

Each use case also is documented with a flow of events. The flow of events for a use case is a description of the events needed to accomplish the required behavior of the use case. The flow of events is written in terms of *what* the system should do, not *how* the system does it. That is, it is written in the language of the domain, not in terms of implementation.

The flow of events should include:

• When and how the use case starts and ends

• What interaction the use case has with the actors

• What data is needed by the use case

• The normal sequence of events for the use case

• The description of any alternate or exceptional flows

The flow of events documentation typically is created in the Elaboration Phase in an iterative manner. At first, only a brief description of the steps needed to carry out the normal flow of the use case (i.e., what functionality is provided by the use case) is written. As analysis progresses, the steps are fleshed out to add more detail. Finally, the exceptional flows are added to the use case (what happens if... part of the flow of events). Each project should use a standard template for the creation of the flow of events document. I have found the following template to be useful.

X Flow of Events for the <name> Use Case

X.1 Preconditions

X.2 Main Flow

X.3 Subflows (if applicable)

X.4 Alternative Flows

Where X is a number from 1 to the number of use cases.

Linking flow of Events Documents to Use Cases in Rational Rose

1. Right-click on the use case in the browser to make the shortcut menu visible.
2. Select the Open Specification menu option.
3. Select the Files tab.
4. Right-click to make the shortcut menu visible.
5. Select the Insert File menu option.
6. Browse to the appropriate directory and select the desired file.
7. Click the Open button.
8. Click the OK button to close the specification.

Linked documents are also added to the Browser.
A linked use case flow of events document is shown in Figure 3-7

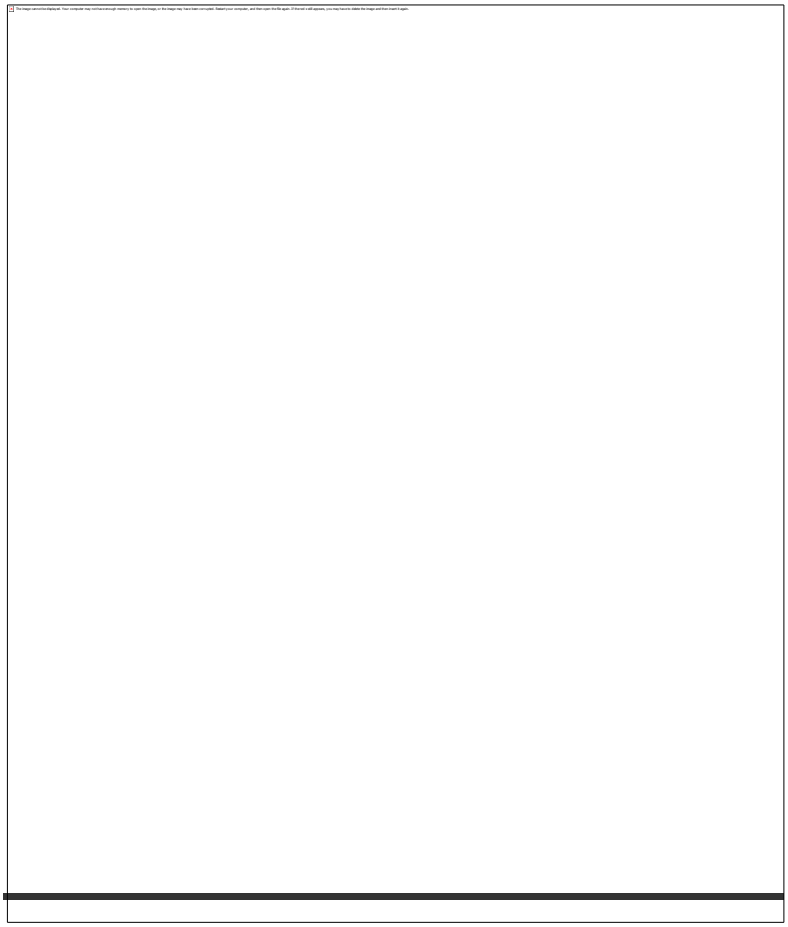


Figure 3-7. Linked Flow of Events Document

Use Case Relationships

An association relationship may exist between an actor and a use case. This type of association is often referred to as a *communicate association* since it represents communication between an actor and a use case. An association may be navigable in both directions (actor to use case and use case to actor) or it may be navigable in only one direction (actor to use case or use case to actor). The navigation direction of an association represents who is initiating the communication (i.e., the actor is initiating the communication with the use case; the use case is initiating the communication with the actor). An association is represented as a line connecting the related

elements. Navigation in only one direction is depicted by adding an arrowhead to the association line that denotes the direction.

Types of Relationships

There are two types of relationships that may exist between use cases: *include* and *extend.* Multiple use cases may share pieces of the same functionality. This functionality is placed in a separate use case rather than documenting it in every use case that needs it. *Include* relationships are created between the new use case and any other use case that "uses" its functionality. For example, each use case in the ESU Course Registration System starts with the verification of the user. This functionality can be captured in a User Verification use case, which is then used by other use cases as needed. An *include* relationship is drawn as a dependency relationship that points from the base use case to the used use case.

An *extend* relationship is used to show

• Optional behavior
• Behavior that is run only under certain conditions such as triggering an alarm
• Several different flows that may be run based on actor selection

For example, a use case that monitors the flow of packages on a conveyer belt can be extended by a Trigger Alarm use case if the packages jam. At this time, no extensions have been identified for the ESU Course Registration System. An *extend* relationship is drawn as a dependency relationship that points from the extension to the base use case. The UML has a concept called a *stereotype,* which provides the capability of extending the basic modeling elements to create new elements. Thus, the concept of a stereotype allows the UML to have a minimal set of symbols that may be extended where needed to provide the communication artifacts that have meaning for the system under development. Stereotype names are included within guillemets (< < > >) and placed along the relationship line. Stereotypes are used to create the needed use case relationships. The stereotype < < communicate> > may be added to an association to show that the association is a communicate association. This is optional since an association is the only type of relationship allowed between an actor and a use case. Include and extend relationships must use stereotypes since they are both represented by a dependency relationship.

Use case relationships are shown in Figure 3-8.

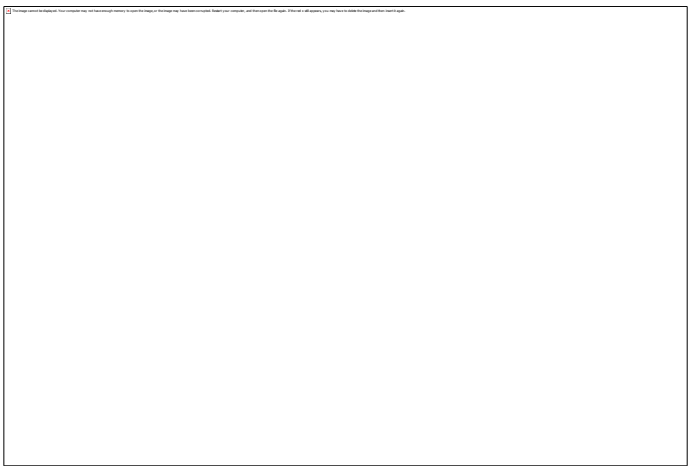


Figure 3-8. Use Case Relationships

Unit-2 Lecture 12

Use Case Diagrams

A Use case diagram is a graphical view of some or all of the actors, use cases, and their interactions identified for a system. Each system typically has a Main Use Case diagram, which is a picture of the system boundary (actors) and the major functionality provided by the system (use cases). Other use case diagrams may be created as needed. Some examples follow:

• A diagram showing all the use cases for a selected actor

• A diagram showing all the use cases being implemented in an iteration

• A diagram showing a use case and all its relationships

Creating The Main Use Case Diagram in Rational Rose

1. Double-click on the Main diagram in the Use Case View in the browser to open the diagram.
2. Click to select an actor in the browser and drag the actor onto the diagram.
3. Repeat step 2 for each additional actor needed in the diagram.
4. Click to select a use case in the browser and drag the use case onto the diagram.
5. Repeat step 4 for each additional use case needed in the diagram.

Note: Actors and use cases may also be created directly on a use case diagram by using the toolbar.

Creating Communicate Associations in Rational Rose

1. Click to select the Association icon or the Unidirectional Association icon from the diagram toolbar. **Note: If the Association icon is not present on the toolbar, it may be added by right-clicking on the toolbar, selecting the Customize menu choice from the shortcut menu, and adding the icon to the toolbar.**
2. Click on an actor initiating a communication and drag the association line to the desired use case.

To add the communicate stereotype (optional):

1. Double-click on the association line to make the Specification visible.

2. Click the arrow in the Stereotype field to make the drop-down menu visible, and select communicate.
3. Click the OK button to close the Specification.
4. Repeat the preceding steps for each additional communicate relationship.

Creating Include Relationships in Rational Rose

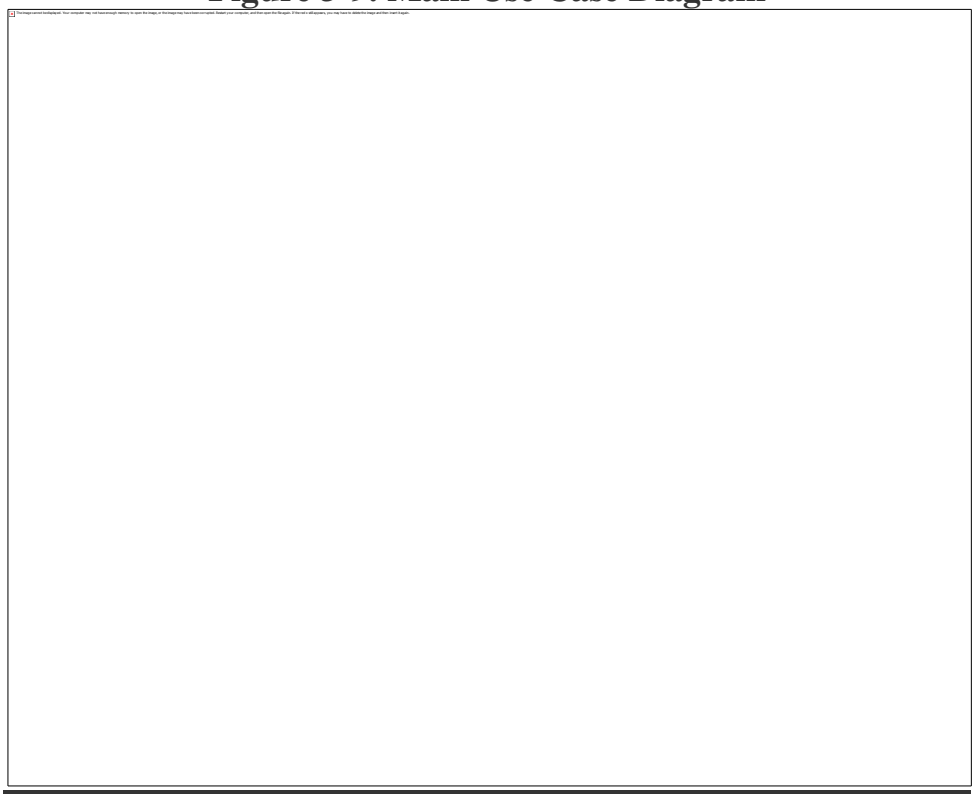
1. Click to select the Unidirectional Association icon from the toolbar.
2. Click on the using use case and drag the Unidirectional Association icon to the used use case.
3. Double-click on the unidirectional association arrow to make the Specification visible.
4. Click the arrow in the Stereotype field to make the drop-down menu visible, and select include.
5. Click the OK button to close the Specification.

Creating Extended Relationships in Rational Rose

1. Click to select the Unidirectional Association icon from the toolbar.
2. Click on the use case containing the extended functionality and drag the Unidirectional Association icon to the base use case.
3. Double-click on the unidirectional association arrow to make the Specification visible.
4. Click the arrow in the Stereotype field to make the drop-down menu visible and select extend.
5. Click the OK button to close the Specification.

The Main use case diagram for the ESU Course Registration System is shown in Figure 3-9.

Figure 3-9. Main Use Case Diagram



Unit-2 Lecture 13

Creating Additional Use Case Diagram in Rational Rose

1. Right-click on the Use Case View in the browser to make the shortcut menu visible.
2. Select the New: Use Case Diagram menu option.
3. While the use case diagram is selected, enter the name of the actor.
4. Open the diagram and add actors, use cases, and interactions to the diagram as needed

An additional use case diagram is shown in Figure 3-10.

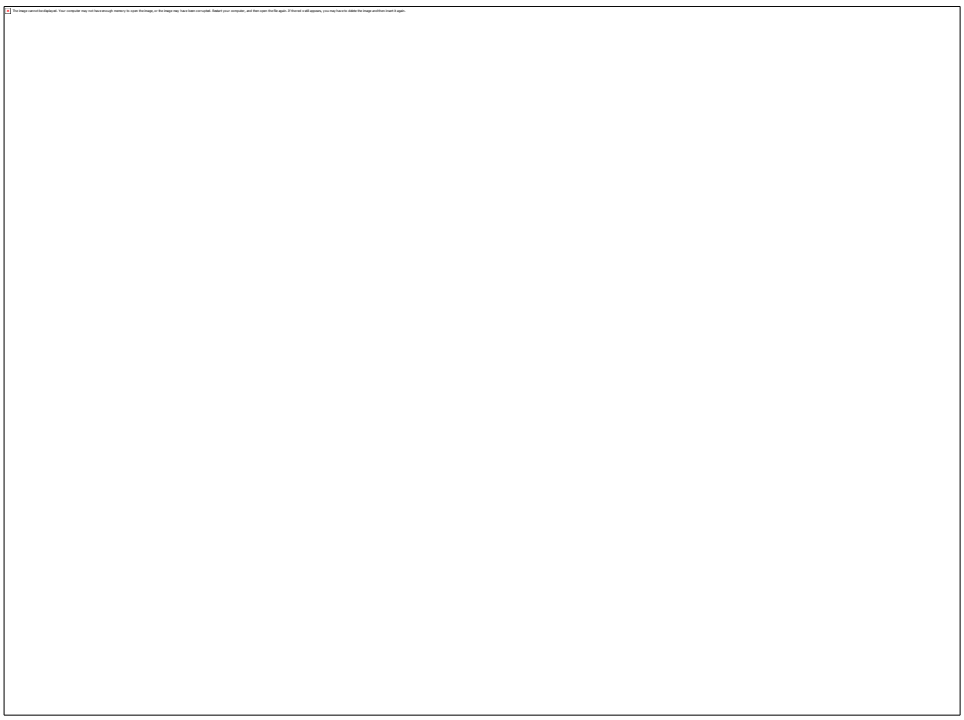


Figure 3-10. An Additional Use Case Diagram

Unit-2 Lecture 14

Activity Diagrams

Activity Diagrams also may be created at this stage in the life cycle. These diagrams represent the dynamics of the system. They are flow charts that are used to show the workflow of a system; that is, they show the flow of control from activity to activity in the system, what activities can be done in parallel, and any alternate paths through the flow. At this point in the life cycle, activity diagrams may be created to represent the flow across use cases or they may be created to represent the flow within a particular use case. Later in the life cycle, activity diagrams may be created to show the workflow for an operation. Activity diagrams contain activities,

transitions between the activities, decision points, and synchronization bars. In the UML, activities are represented as rectangles with rounded edges, transitions are drawn as directed arrows, decision points are shown as diamonds, and synchronization bars are drawn as thick horizontal or vertical bars as shown in Figure 3-11.

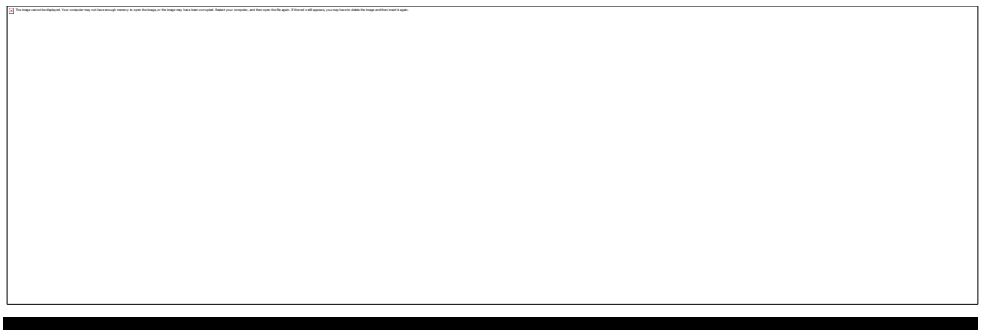


Figure 3-11. UML Notation for Activity Diagram Elements

Creating Activity Diagrams in Rational Rose

1. Right-click on the Use Case View in the browser to make the shortcut menu visible.
2. Select the New: Activity Diagram menu choice. This will add an activity diagram called New Diagram to the browser.
3. While the new diagram is still selected, enter the name of the diagram.
4. Double-click on the activity diagram in the browser to open the diagram.

A browser view of an activity diagram is shown in Figure 3-12.

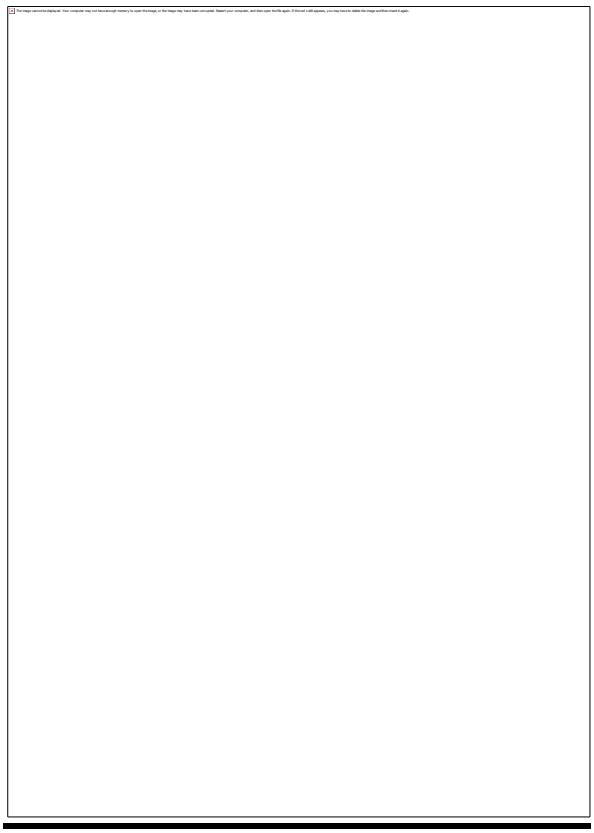


Figure 3-12. Activity Diagram in the Browser

Activities

An activity represents the performance of some behavior in the workflow.

Creating Activities in Rational Rose

1. Click to select the Activity icon from the toolbar.
2. Click on the activity diagram window to place the activity.
3. While the activity is still selected, enter the name of the activity.

Activities are shown in Figure 3-13.

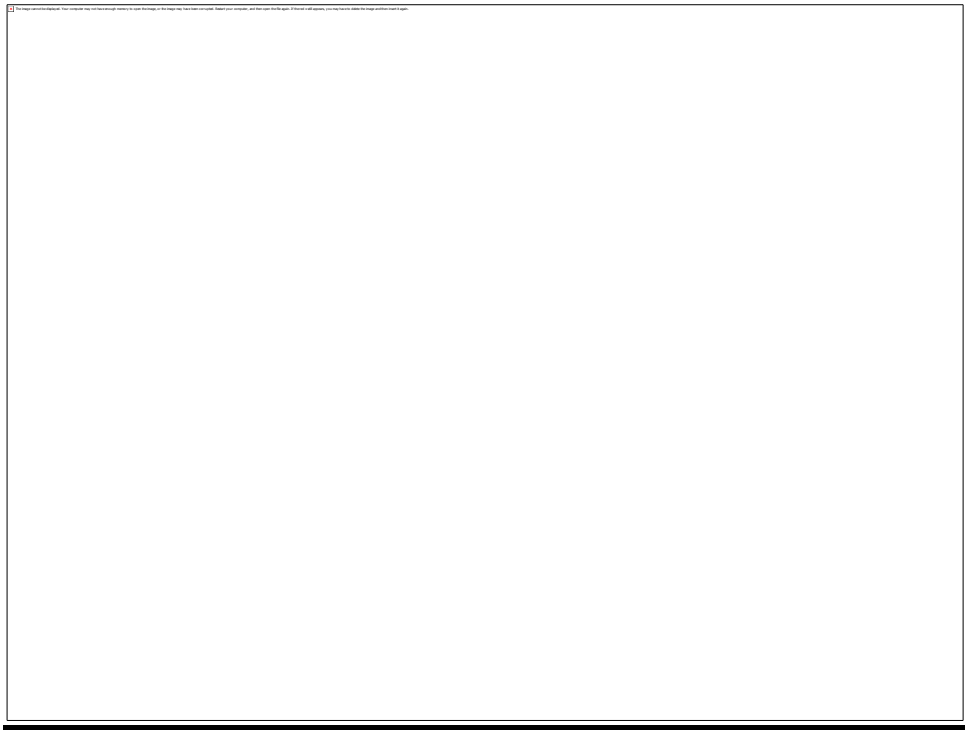


Figure 3-13. Activities

Transitions

Transitions are used to show the passing of the flow of control from activity to activity. They are typically triggered by the completion of the behavior in the originating activity.

Creating Transitions in Rational Rose

1. Click to select the state transition icon from the toolbar.
2. Click on the originating activity and drag the transition arrow to the successor activity.

Transitions are shown in Figure 3-14.

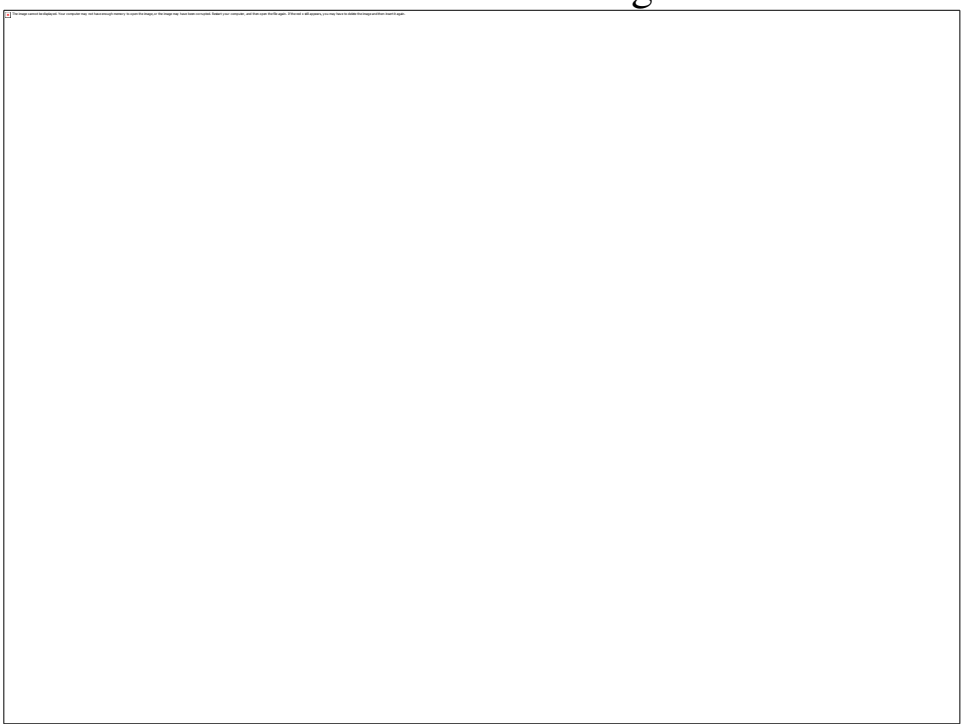


Figure 3-14. Transitions

Decision Points

When modeling the workflow of a system it is often necessary to show where the flow of control branches based on a decision point. The transitions from a decision point contain a guard condition, which is used to determine which path from the decision point is taken. Decisions along with their guard conditions allow you to show alternate paths through a work flow.

Creating Decision Points in Rational Rose

1. Click to select the Decision icon from the toolbar.
2. Click on the activity diagram window to place the decision.
3. While the decision is still selected, enter the name of the decision.
4. Click to select the Transition icon on the toolbar.
5. Click on the originating activity and drag the transition to the decision icon.

A decision point is shown in Figure 3-15.

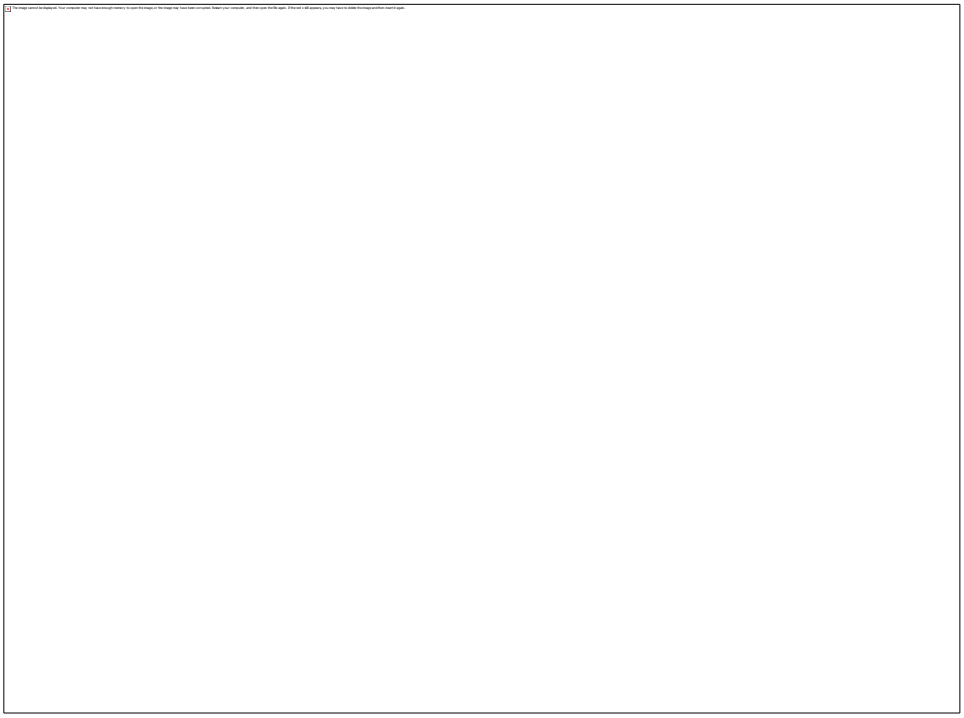


Figure 3-15. Decision in an Activity Diagram

Creating Guarded Transitions in Rational Rose

1. Click to select the State Transition icon from the toolbar.

2. Click on the decision and drag the transition to the successor activity. (Note: Rational Rose may place the transition on top of an existing transition. To separate the transition, select the transition and drag it onto the activity diagram window.)
3. Double-click on the transition arrow to make the Specification visible.
4. Select the Detail tab.
5. Enter the guard condition in the Guard Condition field.
6. Click the OK button to close the Specification.

A transition with a guard is shown in Figure 3-16.

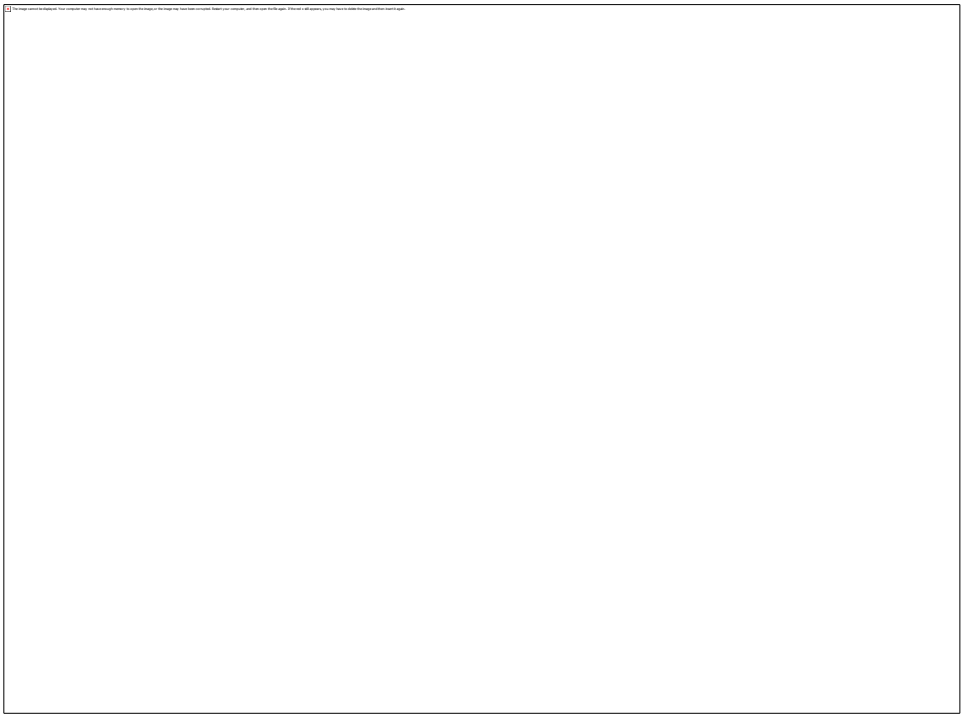


Figure 3-16. Guarded Transition

Creating Rectilinear Lines in Rational Rose

1. Click to select the line that should be rectilinear (multi-select may be accomplished by first selecting the Shift button).
2. Select the Format: Line Style: Rectilinear menu choice.
3. Relocate the lines as needed by selecting the line and dragging it to the desired location on the activity diagram window.

Rectilinear lines are shown in Figure 3-17.

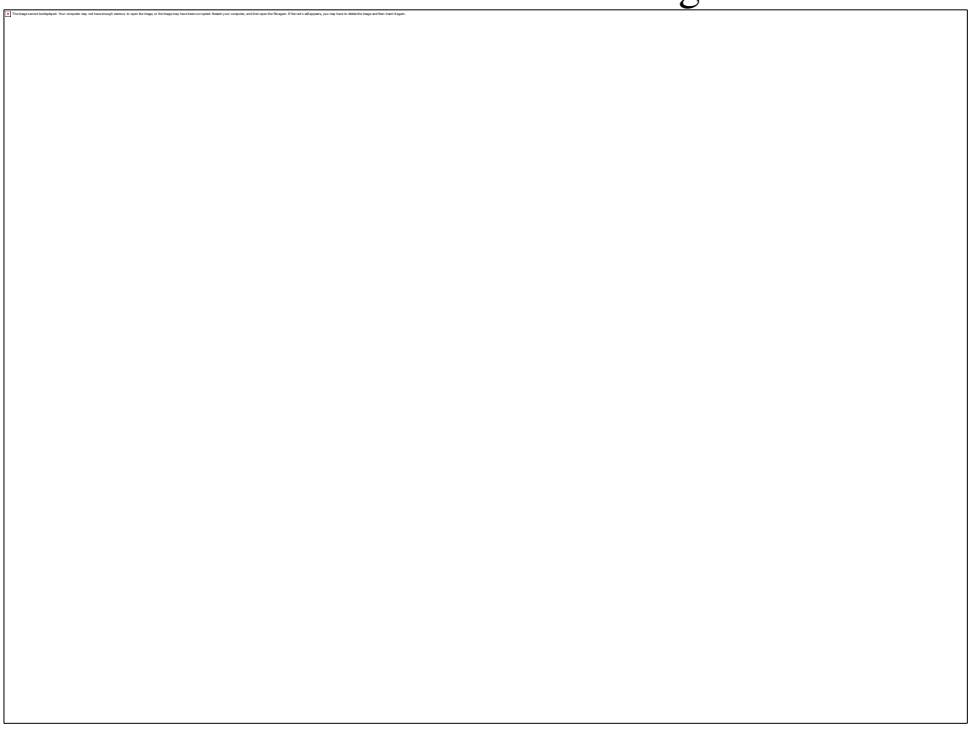


Figure 3-17. Rectilinear Lines

Synchronization Bars

In a workflow there are typically some activities that may be done in parallel. A synchronization bar allows you to specify what activities may be done concurrently. Synchronization bars are also used to show joins in the workflow; that is, what activities must complete before processing may continue. That said, a synchronization bar may have many incoming transitions and one outgoing transition, or one incoming transition and many outgoing transitions.

Creating Synchronization Bars in Rational Rose

1. Click to select the Horizontal Synchronization or the Vertical Synchronization icon from the toolbar.
2. Click on the activity diagram window to place the synchronization bar.
3. Click to select the State Transition icon on the toolbar and add any needed incoming and outgoing transitions to the synchronization bar.

Synchronization bars are shown in Figure 3-18.

Figure 3-18. Synchronization Bars

Swim-lanes

Swim-lanes may be used to partition an activity diagram. This typically is done to show what person or organization is responsible for the activities contained in the swim-lane.

1. Click to select the Swim-lane icon from the toolbar.
2. Click on the activity diagram window to place the swim-lane. This will add a swim-lane called New Swim-lane to the diagram.
3. Double-click on the New Swim-lane (the words) to open the Specification.
4. Enter the name of the swim-lane in the Name field.
5. Click the OK button to close the Specification.

6. To resize the swim-lane, click on the swim-lane border and drag the swim-lane to the desired location.

7. Drag all needed activities and transitions into the swim-lane. (Note: You may also create new activities and transitions in the swim-lane.)

An activity diagram with swim-lanes is shown in Figure 3-19.

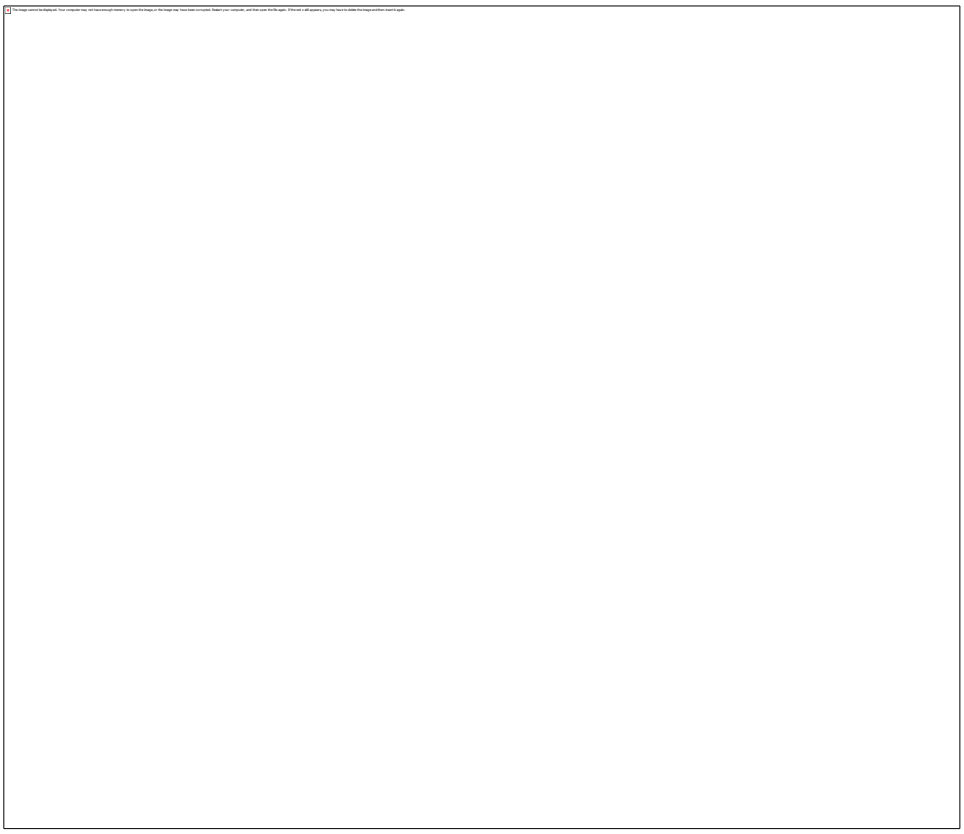


Figure 3-19. Swim-lanes

Unit-3 Lecture 15

Identifying Classes , Packages and drawing a Class Diagram

What is an Object?

An object is a representation of an entity, either real-world or conceptual. An object can represent something concrete, such as Joe's truck or my computer, or a concept such as a chemical process, a bank transaction, a purchase order, Mary's credit history, or an interest rate.

An object is a concept, abstraction, or thing with well-defined boundaries and meaning for an application. Each object in a system has three characteristics: state, behavior, and identity.

State, Behavior, and Identity

The state of an object is one of the possible conditions in which it may exist. The state of an object typically changes over time, and is defined by a set of properties (called attributes), with the values of the properties, plus the relationships the object may have with other objects. For example, a course offering object in the registration system may be in one of two states: *open* and *closed*. If the number of students registered for a course offering is less than 10, the state of the course offering is open. When the tenth student registers for the course offering, the state becomes closed.

Behavior determines how an object responds to requests from other objects and typifies everything the object can do. Behavior is implemented by the set of operations for the object. In the registration system, a course offering could have the behaviors *add a student* and *delete a student*. Identity means that each object is unique—even if its state is identical to that of another object. For example, Algebra 101, Section 1, and Algebra 101, Section 2 is two objects in the Course Registration System. Although they are both course offerings, they each have a unique identity.

In the UML, objects are represented as rectangles and the name of the object is underlined as shown in Figure 4-1.

Figure 4-1. UML Notation for an Object

What is a Class?

A class is a description of a group of objects with common properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. Thus, a class is a template to create objects. Each object is an instance of some class and objects cannot be instances of more than one class. For example, the Course Offering class may be defined with the following characteristics:

• Attributes—location, time offered
• Operations—retrieve location, retrieve time of day, add a student to the offering

Algebra 101, Section 1, and Algebra 101, Section 2 are objects belonging to the Course-Offering class. Each object would have a value for the attributes and access to the operations specified by the Course-Offering class.

A good class captures one and only one abstraction—it should have one major theme. For example, a class that has the capability of maintaining information about a student and the information about all the course offerings that the student has taken over the years is not a good class since it does not have one major theme. This class should be split into two related classes: Student and Student History.

Classes should be named using the vocabulary of the domain. The name should be a singular noun that best characterizes the abstraction. Acronyms may be used if the acronym has the same meaning for all involved, but if an acronym has different meanings for different people then the full name should always be used. If a class is named with an acronym, the full name should also be contained in the class documentation.

It is often hard to distinguish between an object and a class. Why is Algebra 101, Section 1 an object and not a class? What makes it different from Algebra 101, Section

2? The answers to these questions are very subjective. By looking at their structure and behavior, it can be seen that both have the same structure and behavior. They are only different course offerings for a semester. In addition, it may be noted that there are many other "things" in the Course Registration System that have the same structure and behavior (e.g., Music 101, Section1; History 101, Section 1; and History 101, Section 2). This leads to the decision to create a Course-Offering class.

In the UML, classes are represented as compartmentalized rectangles. The top compartment contains the name of the class, the middle compartment contains the structure of the class (attributes), and the bottom compartment contains the behavior of the class (operations). **A class is shown in Figure 4-2**.

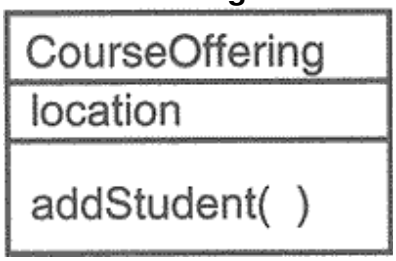


Figure 4-2. UML Notation for a Class

Creating Classes in Rose Browser

1. Right-click to select the Logical View in the browser.
2. Select the New: Class menu choice. A class called New Class is placed in the browser.
3. While the new class is still selected, enter the name of the class.

The browser view of a class is shown in Figure 4-3.

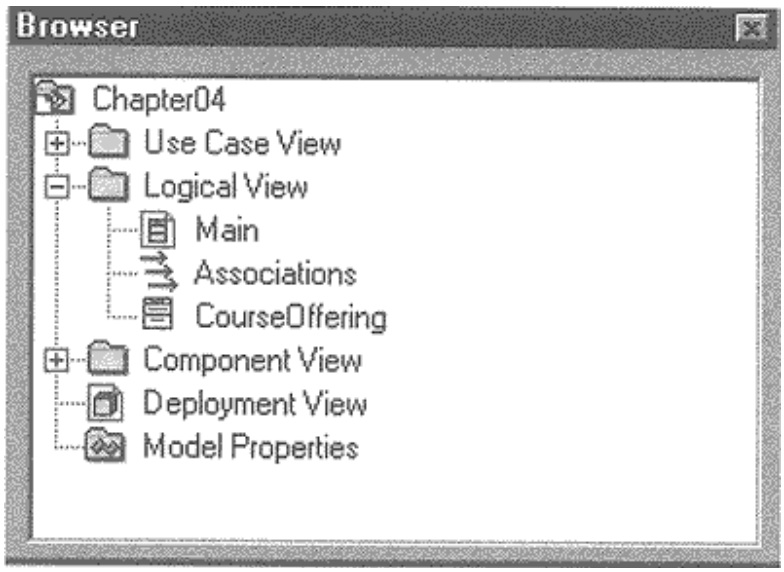


Figure 4-3. Class Created in the Browser

Stereotypes and Classes

We previously talked about stereotypes for relationships in use case diagrams. Classes can also have stereotypes. As before, a stereotype provides the capability to create a new kind of modeling element. Here, we can create new kinds of classes. Some common stereotypes for a class are entity, boundary, control, utility, and exception.

The stereotype for a class is shown below the class name enclosed in guillemets (<< >>). If desired, a graphic icon or a specific color may be associated with a stereotype. In Rational Rose 2000, icons for the Rational Unified Process stereotypes of control, entity, and boundary are supplied. These stereotypes are shown in Figure 4-4 along with an example of a class with a stereotype of exception.

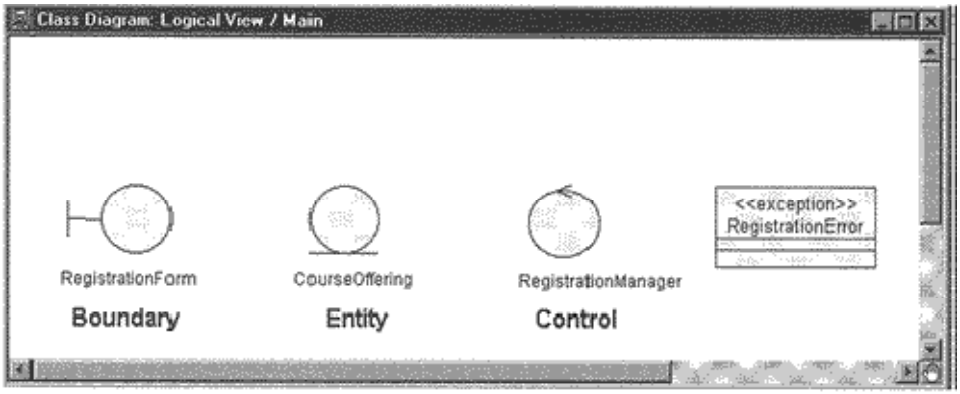


Figure 4-4. Classes with Stereotypes

Discovering Classes

A cookbook for finding classes does not exist. As Grady Booch has been known to say, "This is hard!" The Rational Unified Process advocates finding the classes for a system under development by looking for boundary, control, and entity classes. These three stereotypes conform to a "model-view-controller" point of view and allow the analyst to partition the system by separating the view from the domain from the control needed by the system.

Since the analysis and design process is iterative, the list of classes will change as time moves on. The initial set of classes probably will not be the set of classes that eventually gets implemented. Thus, the term candidate class is often used to describe the first set of classes found for a system.

Entity Classes

An entity class models information and associated behavior that is generally long lived. This type of class may reflect a real-world entity or it may be needed to perform tasks internal to the system. They are typically independent of their surroundings; that is, they are not sensitive to how the surroundings communicate with the system. Many times, they are application independent, meaning that they may be used in more than one application.

The first step is to examine the responsibilities documented in the flow of events for the identified use cases (i.e., what the system must do). Entity classes typically are classes that are needed by the system to accomplish some responsibility. The nouns and noun phrases used to describe the responsibility may be a good starting point. The initial list of nouns must be filtered because it could contain nouns that are outside the problem domain, nouns that are just language expressions, nouns that are redundant, and nouns that are descriptions of class structures.

Entity classes typically are found early in the Elaboration Phase. They are often called "domain" classes since they usually deal with abstractions of real-world entities.

Boundary Classes

Boundary classes handle the communication between the system surroundings and the inside of the system. They can provide the interface to a user or another system (i.e., the interface to an actor). They constitute the surroundings-dependent part of the system. Boundary classes are used to model the system interfaces.

Each physical actor/scenario pair is examined to discover boundary classes. The boundary classes chosen in the Elaboration Phase of development are typically at a high level. For example, you may model a window but not model each of its dialogue boxes and buttons. At this point, you are documenting the user interface requirements, not implementing the interface.

User interface requirements tend to be very vague—the terms user-friendly and flexible seem to be used a lot. But user-friendly means different things to different people. This is where prototyping and storyboarding techniques can be very useful. The customer can get the "look and feel" of the system and truly capture *what* user-friendly means. What is then captured as the structure and behavior of the boundary class. During design these classes are refined to take into consideration the chosen user interface mechanisms—how they are to be implemented.

Boundary classes are also added to facilitate communication with other systems. During design, these classes are refined to take into consideration the chosen communication protocols.

Control Classes

Control classes model sequencing behavior specific to one or more use cases. Control classes coordinate the events needed to realize the behavior specified in the use case.

You can think of a control class as "running" or "executing" the use case—they represent the dynamics of the use case. Control classes typically are application-dependent classes.

In the early stages of the Elaboration Phase, a control class is added for each actor/use case pair. The control class is responsible for the flow of events in the use case.

The use of control classes is very subjective. Many authors feel that the use of control classes results in behavior being separated from data. This can happen if your control classes are not chosen wisely. If a control class is doing more than sequencing, then it is doing too much! For example, in the Course Registration System, a student selects course offerings and if the course offering is available, the student is added to it. Who knows how to add the student—the control class or the course offering? The right answer is the course offering. The control class knows when the student should be added; the course offering knows how to add the student. A bad control class would not only know when to add the student but how to add the student.

The addition of a control class per actor/use case pair is only an initial cut—as analysis and design continues, control classes may be eliminated, split up, or combined.

Creating Stereotypes For Classes in Rational Rose

1. Right-click to select the class in the browser and make the shortcut menu visible.
2. Select the Open Specification menu choice.
3. Select the General tab.
4. Click the arrow in the Stereotype field to make the drop-down menu visible and select the desired stereotype or, to create a new stereotype, enter the name of the stereotype in the Stereotype field.
5. Click the OK button to close the Specification.

The Specification for the Student class is shown in Figure 4-5. If the default language for a model is set to a language other than analysis (Tools: Options menu choice, Notation tab), then a tab for that language will be added to the class specification.

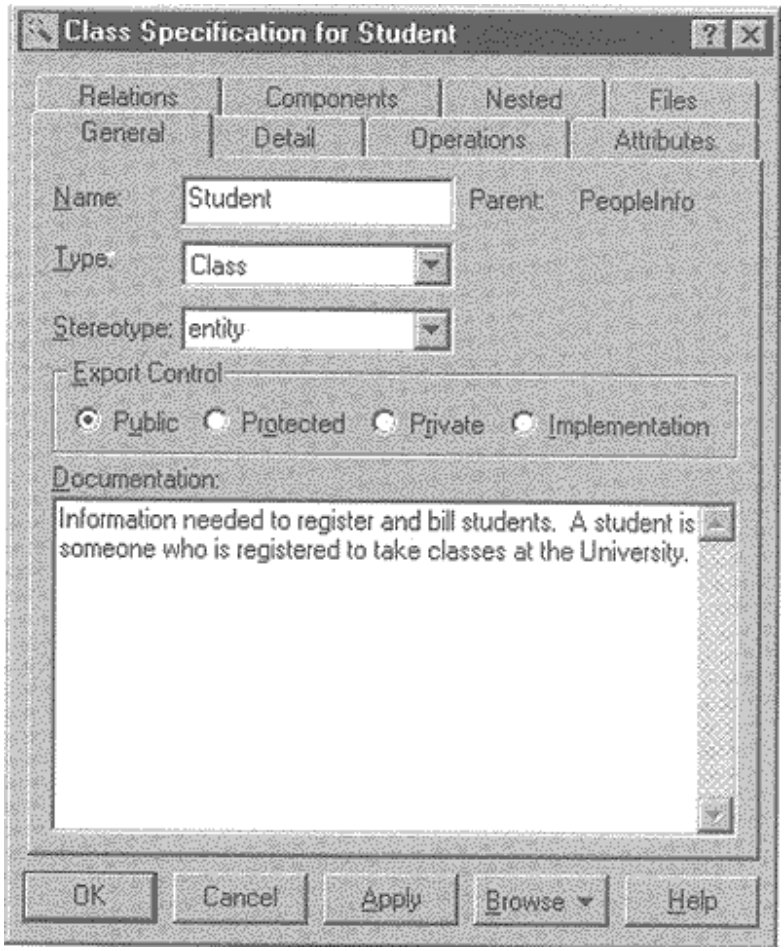


Figure 4-5. Setting a Class Stereotype

Unit-3 Lecture 16

Documenting Classes

As classes are created, they should also be documented. The documentation should state the purpose of the class and not the structure of the class. For example, a Student class could be documented as follows:

Information needed to register and bill students. A student is someone currently registered to take classes at the University.

A bad definition would be the following:

The name, address, and phone number of a student.

This definition only tells me the structure of the class, which can be determined by looking at its attributes. It does not tell me why I need the class.

Difficulty in naming or documenting a class may be an indication that it is not a good abstraction. The following list typifies things that can happen as classes are named and documented:

• Can identify a name and a clear concise definition—good candidate class
• Can identify a name, but the definition is the same as another class—combine the classes
• Can identify a name, but need a book to document the purpose—break up the class
• Cannot identify a name or a definition—more analysis is needed to determine the correct abstractions

Documenting Classes in Rational Rose

1. Click to select the class in the browser.
2. Position the cursor in the documentation window and enter the documentation for the class.

The description of the Student class is shown in Figure 4-6.

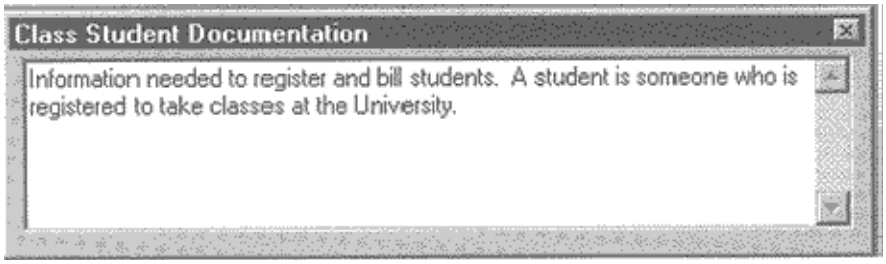

Class Student Documentation
Information needed to register and bill students. A student is someone who is registered to take classes at the University.

Figure 4-6. Class Documentation

Packages

If a system contained only a few classes, you could manage them easily. Most systems are composed of many classes, and thus you need a mechanism to group them together for ease of use, maintainability, and reusability. This is where the concept of a package is useful. A package in the logical view of the model is a collection of related packages and/or classes. By grouping classes into packages, we can look at the "higher" level view of the model (i.e., the packages) or we can dig deeper into the model by looking at what is contained by the package.

Each package contains an interface that is realized by its set of public classes—those classes to which classes in other packages talk. The rest of the classes in a package are implementation classes—classes do not communicate with classes in other packages.

If the system is complex, packages may be created early in the Elaboration Phase to facilitate communication. For simpler systems, the classes found early in analysis may be grouped into one package—the system itself. As analysis and design progress, the

concept of a package will be used to group the classes that are needed to carry out the architectural decisions made for the system.

In the UML, packages are represented as folders. A package is shown in Figure 4-7.

Figure 4-7. UML Notation for a Package

Creating Packages in Rose Browser

1. Right-click to select the Logical View in the browser.
2. Select the New: Package menu choice.
3. While the package is still selected, enter the name of the package.

A package created via the browser is shown in Figure 4-8.

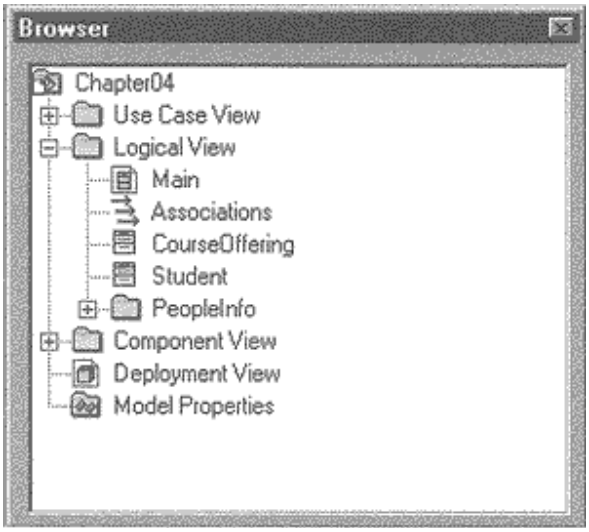


Figure 4-8. Package Created in the Browser

As packages are created, classes in the model are relocated.

Relocating Classes in Rose Browser

1. Click to select the class in the browser.
2. Drag the class to the desired package.
3. Repeat the steps for each class that is to be relocated.

Relocated classes are shown in Figure 4-9.

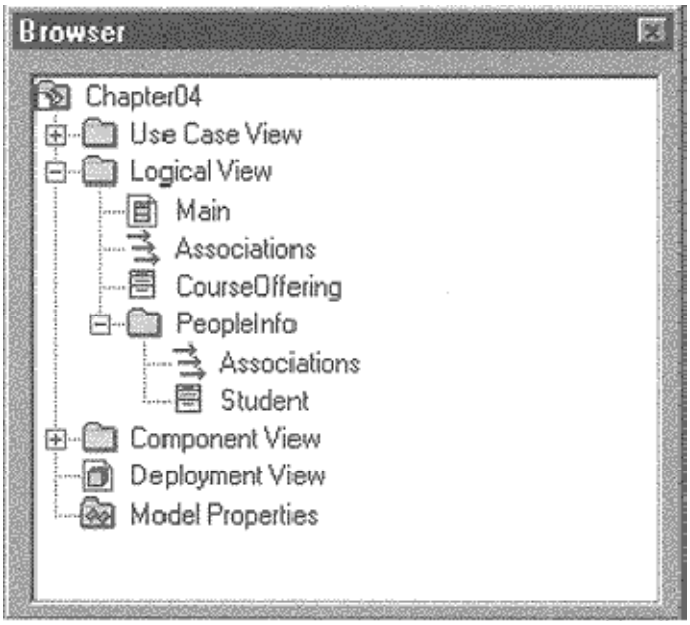


Figure 4-9. Relocated Classes

Unit-3 Lecture 17

Drawing a Class Diagram Specifying Relationships:

The Need of Defining Relationships

All systems are made up of many classes and objects. System behavior is achieved through the collaborations of the objects in the system. For example, a student is added to a course offering when the course offering receives the *add student* message. This is often referred to as an object sending a message to another object. Relationships provide the conduit for object interaction. Two types of relationships discovered during analysis are:-

1. **Associations Relationships** and
2. **Aggregations Relationships**

Unit-3 Lecture 18

Association and Aggregation Relationships, Naming Relationships

Association Relationships

An association is a bidirectional semantic connection between classes. It is not a data flow as defined in structured analysis and design—data may flow in either direction across the association. An association between classes means that there is a link between objects in the associated classes. For example, an association between the Course class and the Professor Course Manager class means that objects in the Course class are connected to objects in the Professor Course Manager class. The number of objects connected depends upon the multiplicity of the association. In the UML, association relationships are shown as a line connecting the associated classes, as shown in Figure 6-1.

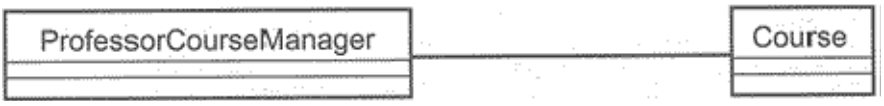


Figure 6-1. UML Notation for an Association Relationship

Creating an **Association Relationship** in Rational Rose

1. Click to select the Association icon from the toolbar. The association icon may be added to the toolbar by right-clicking on the toolbar and selecting the Customize menu command.
2. Click on one of the associated classes in a class diagram.
3. Drag the association line to the other associated class.

An association relationship is shown in figure 6-2

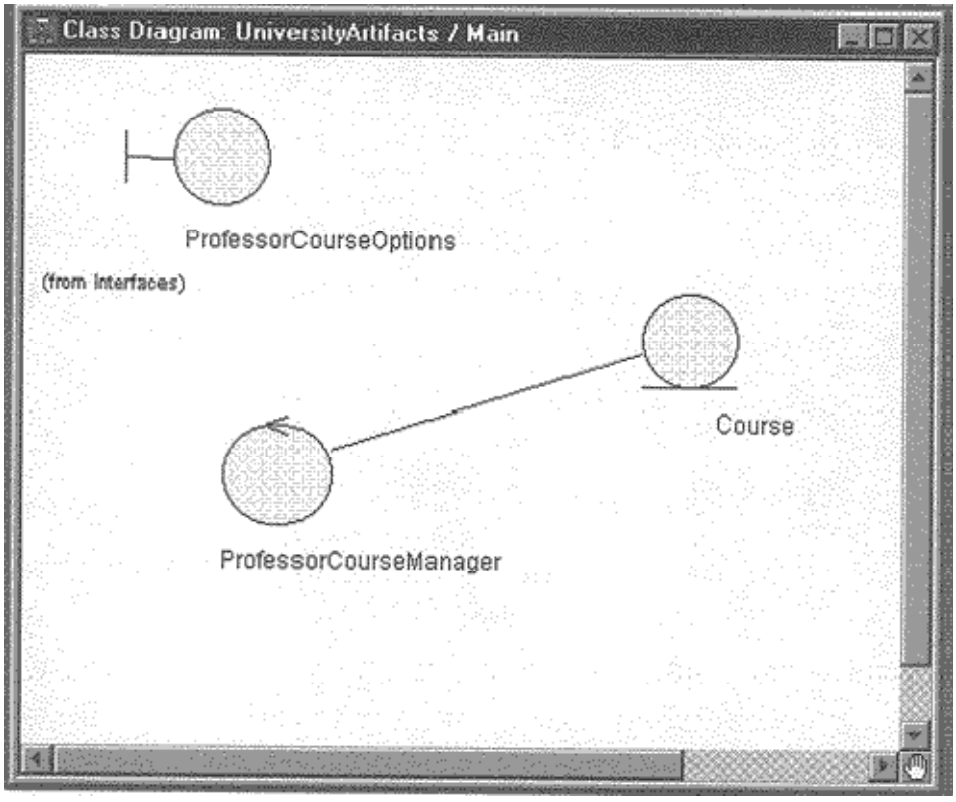


Figure 6-2 Association Relationship

Aggregation Relationships

An aggregation relationship is a specialized form of association in which a whole is related to its part(s). Aggregation is known as a "part-of" or containment relationship.

The UML notation for an aggregation relationship is an association with a diamond next to the class denoting the aggregate (whole), as shown in Figure 6-3.

Figure 6-3 UML Notation for an Aggregation Relationship

Figure 6-3 UML Notation for an Aggregation Relationship

The following tests may be used to determine if an association should be an aggregation:

1. Is the phrase "part of" used to describe the relationship?

2. Are some operations on the whole automatically applied to its parts? For example, delete a course then delete all of its course offerings.

3. Is there an intrinsic asymmetry to the relationship where one class is subordinate to the other?

For example, a Course (Math 101) may be offered at different times during a semester. Each offering is represented as a Course Offering (e.g., Math 101, Section 1, and Math 101, Section 2). The relationship between a Course and a Course Offering is modeled as an aggregation—a Course "has" Course Offerings.

Creating an **Aggregation Relationship** in Rational Rose

1. Select the Aggregation icon from the toolbar. The Aggregation icon may be added to the toolbar by right-clicking on the toolbar and selecting the Customize menu command.
2. Click on the class playing the role of the "whole" in a class diagram and drag the aggregation line to the class playing the role of the "part."

An aggregation relationship is shown in Figure 6-4.

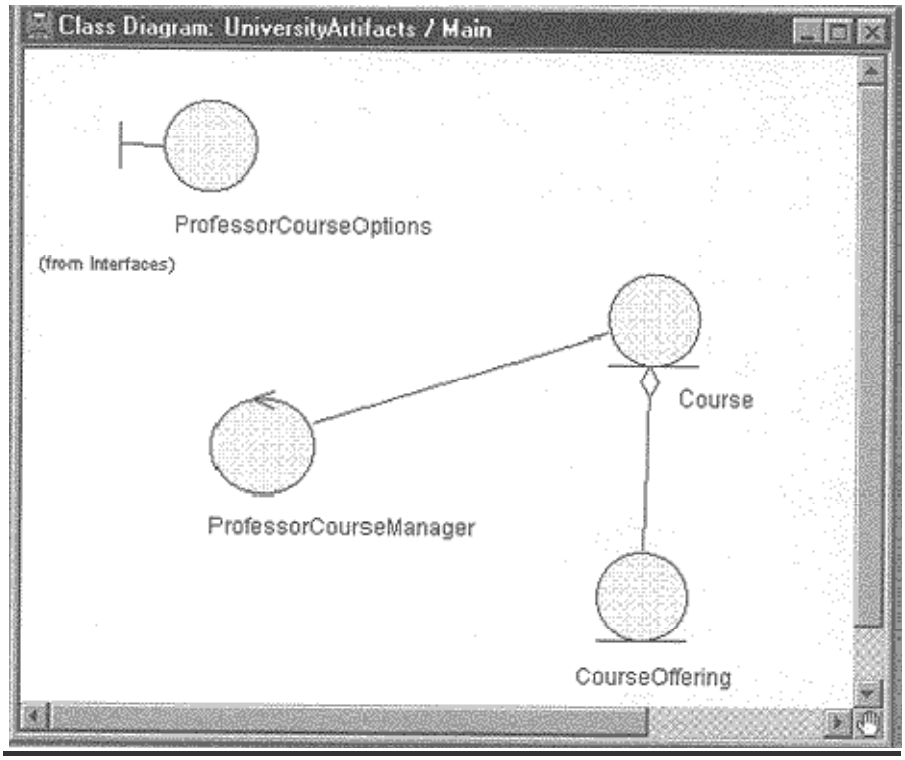


Figure 6-4 Aggregation Relationship

Unit-3 Lecture 19, 20 & 21

Role Names

The end of an association where it connects to a class is called an association role. Role names can be used instead of association names. A role name is a noun that denotes the purpose or capacity wherein one class associates with another. The role name is placed on the association near the class that it modifies, and may be placed on one or both ends of an association line. It is not necessary to have both a role name and an association name.

Creating Role Names in Rational Rose

1. Right-click on the relationship line near the class that it modifies to make the shortcut menu visible.
2. Select the Role Name menu choice.
3. Enter the name of the role.

Figure 6-6 shows an association with a role name.

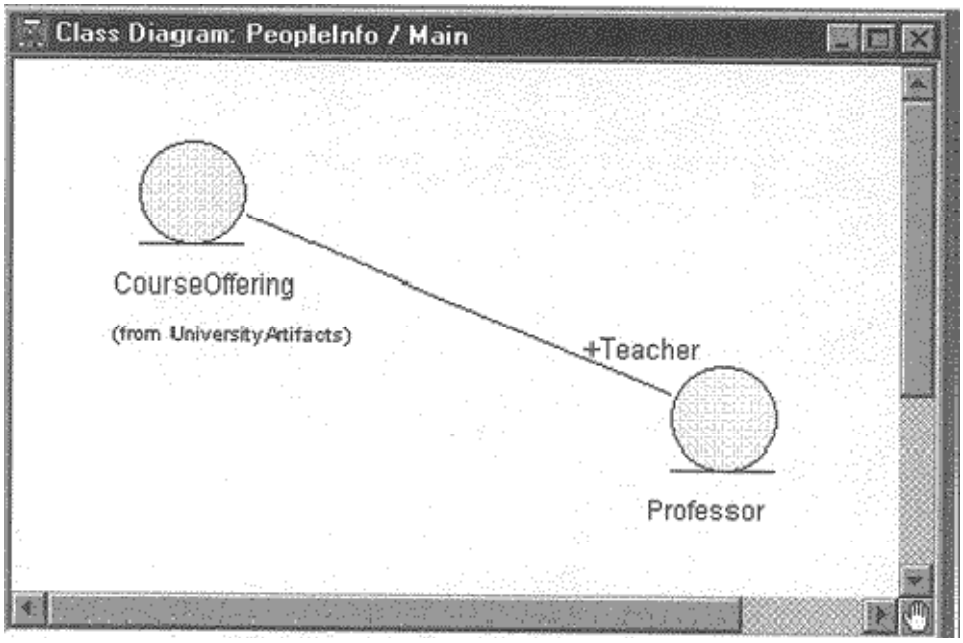


Figure 6-6 Role Name

The relationship shown in Figure 6-6 is read in both directions.

1. A Professor playing the role of the Teacher is related to the Course Offering

2. A Course Offering is related to a Professor playing the role of a Teacher

There is no standard as to whether you should use association names or role names. However, most people today tend to use role names instead of association names since it is easier to convey the meaning of the relationship. This is especially true in a bidirectional relationship since it is very hard to find a verb phrase that reads correctly in both directions. What verb phrase could be used to name the association in Figure 6-6? By using a role name, the meaning is clear in both directions.

Associations are named or role names are used only when the names are needed for clarity. If you have a relationship between Company and Person then you could use an association name called "employs" or the role names of "Employer" and "Employee" to convey an employment relationship. If the classes were named Employer and Employee, no name would be necessary since the meaning of the relationship is clear based on the names of the classes.

Multiplicity Indicators

Although multiplicity is specified for classes, it defines the number of objects that participate in a relationship. Multiplicity defines the number of objects that are linked to one another. There are two multiplicity indicators for each association or aggregation—one at each end of the line.

Some common multiplicity indicators are:-

1 Exactly one

0.. *Zero or more

1.. *One or more

0..1 Zero or one

5..8 Specific range (5, 6, 7, or 8)

4..7,9 Combination (4, 5, 6, 7, or 9)

Creating Multiplicity in Rational Rose

1. Double-click on the relationship line to make the Specification visible.
2. Select the Detail tab for the role being modified (Role A Detail or Role B Detail).
3. Enter the desired multiplicity in the Cardinality field.
4. Click the OK button to close the Specification.

Multiplicity indicators are shown in Figure 6-7.

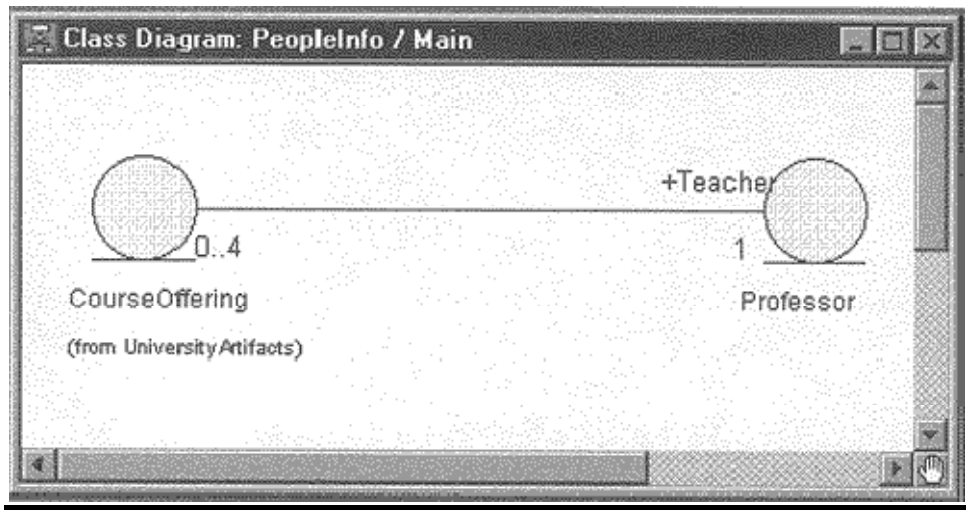


Figure 6-7. Multiplicity Indicators

The drawing in Figure 6-7 may be read in the following ways:

1. One Course Offering object is related to *exactly one* Professor object playing the role of the Teacher. For example, Math 101, Section 1 (a Course Offering object) is related to Professor Smith (a Professor object).

2. One Professor object playing the role of the Teacher is related to *zero to four* Course Offering objects. For example, Professor Smith (a Professor object) is related to Math 101, Section 1; Algebra 200, Section 2; and Calculus 1, Section 3 (Course Offering objects). Since the multiplicity is a range of zero to four, as few as zero Course Offering objects to a maximum of four Course Offering objects may be linked to one Professor Object.

Reflexive Relationships

Multiple objects belonging to the same class may have to communicate with one another. This is shown on the class diagram as a reflexive association or aggregation. Role names rather than association names typically are used for reflexive relationships.

Creating Reflexive Relationship in Rational Rose

1. Select the Association (or Aggregation) icon from the toolbar.
2. Click on the class and drag the association (or aggregation) line outside the class.
3. Release the mouse button.
4. Click and drag the association (or aggregation) line back to the class.
5. Enter the role names and multiplicity for each end of the reflexive association (or aggregation).

A reflexive relationship is shown in Figure 6-8

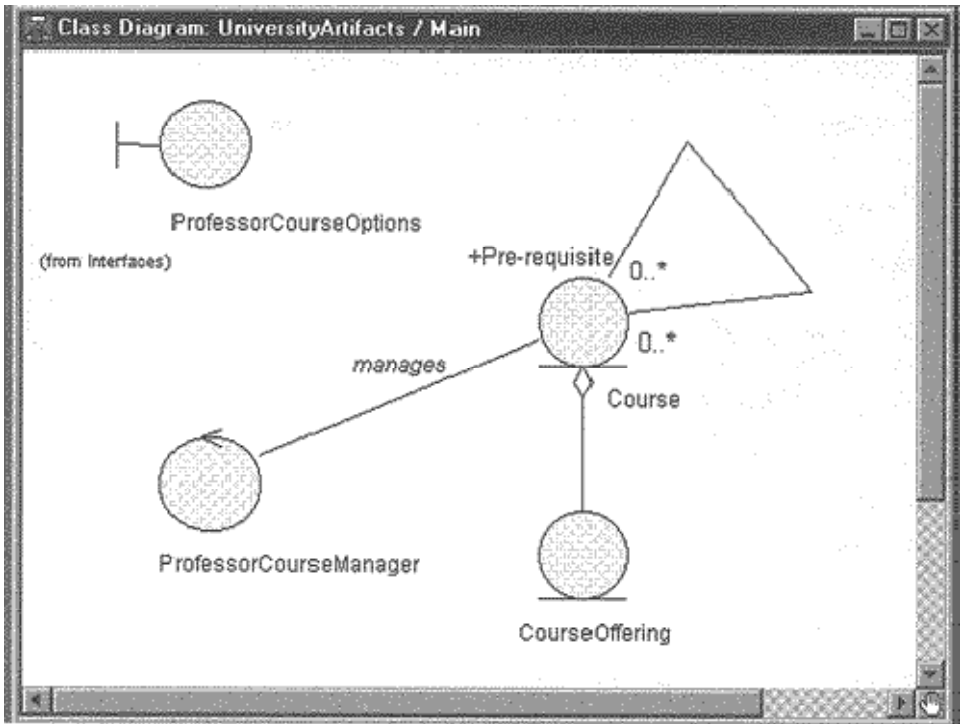


Figure 6-8. Reflexive Relationship

The reflexive relationship in Figure 6-8 may be read in the following ways:

1. One Course object playing the role of Prerequisite is related to zero or more Course objects.

2. One Course object is related to zero or more Course objects playing the role of Prerequisite.

Package Relationships

Package relationships are also added to the model. The type of relationship is a dependency relationship, shown as a dashed arrow to the dependent package, as shown in Figure 6-10. If package A is dependent on package B this implies that one or more classes in package A initiates communication with one or more public classes in package B. Package A is referred to as the Client package and package B is referred to as the Supplier package. Package relationships are also discovered by examining the scenarios and class relationships for the system under development. As this is an iterative process, the relationships will change as analysis and design progresses.

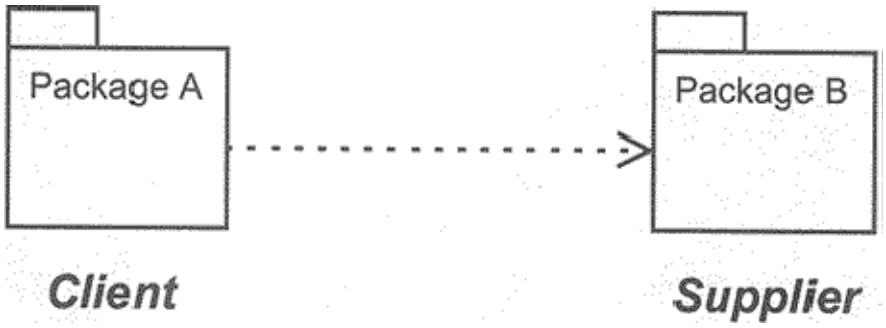


Figure 6-10. Package Relationships

Package Relationships in the ESU Course Registration

In the *Add a Course Offering* scenario, the Add A Course Offering class sends a message to the Professor Course Manager class. This implies that there is a relationship between the Interfaces package and the University Artifacts package. At this time, we have not discovered any relationships to the People package.

Creating Package Relationships in Rational Rose

1. Select the dependency relationship icon from the toolbar.
2. Click on the client package and drag the arrow to the supplier package.

The package relationships for the Course Registration System are shown in Figure 6.11

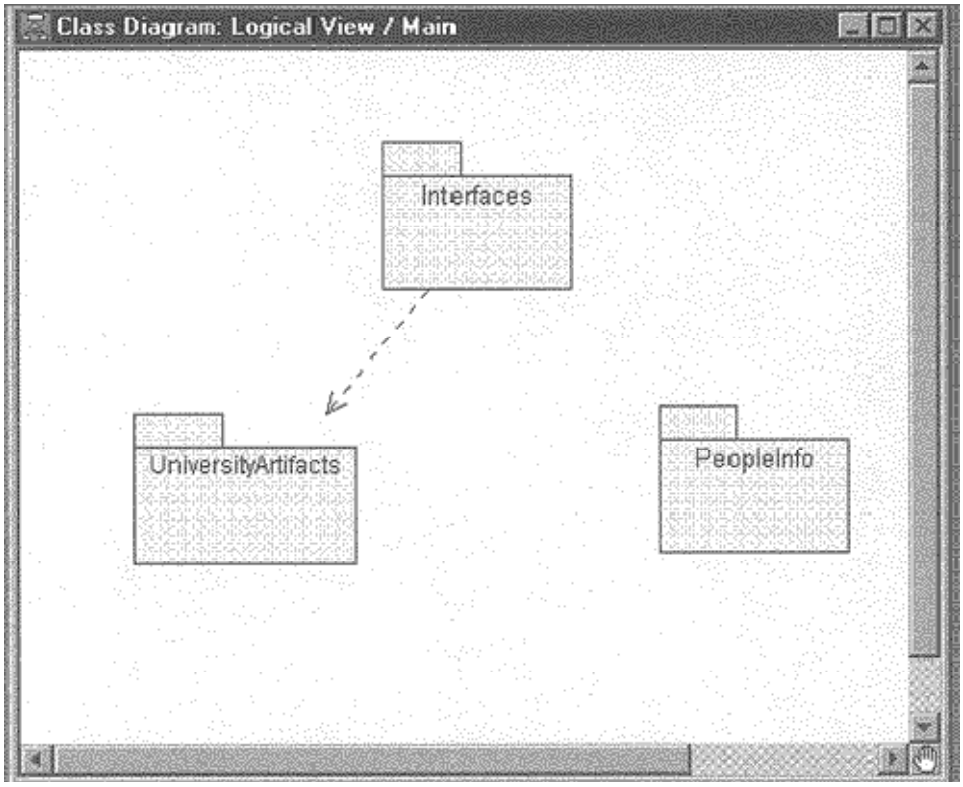


Figure 6-11. Package Relationships in the ESU Course Registration System

Unit-4 Lecture 24

Adding Behavior and Structure

Representing Behavior and Structure

A class embodies a set of responsibilities that define the behavior of the objects in the class. The responsibilities are carried out by the operations defined for the class. An operation should do only one thing and it should do it well! For example, a Course Offering class needs to be able to add a student and remove a student. These responsibilities are represented by two operations—one that knows how to add a student and one that knows how to remove a student. All instances of the class will be able to perform the identified operations of the class.

The structure of an object is described by the attributes of the class. Each attribute is a data definition held by objects of the class. Objects defined for the class have a value for every attribute of the class. For example, a Course class has the attributes of name, definition, and number of credit hours. This implies that every Course object will have a value for each attribute. Attribute values do not have to be unique—there are many three credit courses in the university.

As with classes, style guides should be followed while defining attributes and operations. In this case study, attributes and operations start with a lowercase letter, and underscores are not used—names composed of multiple words are pushed together and the first letter of each additional word is capitalized (e.g., number Of Students). Care should be taken to ensure that appropriate style guides are followed for all defined attributes and operations. This provides consistency across the classes, which leads to more readable and maintainable models and code.

If an object in a class does not need an attribute or operation, look at the class definition. This may be a sign that the class is not cohesive and should be broken up into separate classes. For example, suppose the Course Offering class had the following attributes: offer Number, location, time Of Day, department, number Offerings In Department. A Course Offering may care about its department but it probably does not care about the number of other offerings in the department. A better model would be a Course Offering class related to a Department class. This is in keeping with the general rule that a class should have a major theme.

Unit-4 Lecture 25

Creating Attributes & operations and documenting them

Creating Operations

Messages in interaction diagrams typically are mapped to operations of the receiving class. However, there are some special cases where the message does not become an operation. If the receiving class is a boundary class that is a placeholder for a graphical user interface (GUI) type class, the message is a statement of the requirements for the GUI. These types of messages typically are implemented as some type of GUI control (i.e., a button) and are not mapped to operations since the behavior is carried out by the GUI control itself. For example, the Professor actor must enter a password to start the *Add a Course Offering* scenario. This is represented as a message to the boundary class Professor Course Options. This will never be an operation of the user interface class—it will most likely be a text field on a window. Messages to and from actors also receive special consideration. If the message is to or from an actor that represents a physical person, the message is a statement of human procedure and is therefore incorporated into a user manual, but not to an operation, since it does not make sense to create operations for humans. In the *Add a Course Offering* scenario, the fact that the Professor must have some sort of password to activate the system is an important requirement that should be captured in the user manual. If the message is to or from an actor that represents an external system, a class is created to hold the protocol that is

used to perform the communication with the external system. In this case, the message is mapped to an operation of the class.

Operations should be named in terms of the class performing the operation, not the class asking for the functionality to be performed. For example, the operation to add a student to a course offering is called *add Student().* Additionally, operation names should not reflect the implementation of the operation since the implementation may change over time. For example, each Course Offering object has a maximum of ten students assigned to it. You may need to ask the Course Offering object how many students are assigned to it at a point in time. Today, the number is calculated by looking at the links from the Course Offering object to Student objects. An operation called *calculate Number of Students ()* implies there is a calculation involved. Although this is the case today, it may not always be true (next year, the implementation may change to store the value in a file). A better name for the operation is *get Number of Students ()* since this does not imply how the operation is implemented.

Mapping Messages to New Operations in Rational Rose

1. Assign the objects to classes if that has not been done previously.
2. Right-click on the message arrow to make the shortcut menu visible.
3. Select the <new operation> menu choice. This will open the Operation Specification.
4. Enter the name of the operation in the Operation Specification.
5. Click the OK button to close the Operation Specification.

Note: If the desired operation already exists for the class, you do not have to recreate the operation; simply select the operation from the list of operations for the class.

A sequence diagram with operations is shown in Figure 7-1.

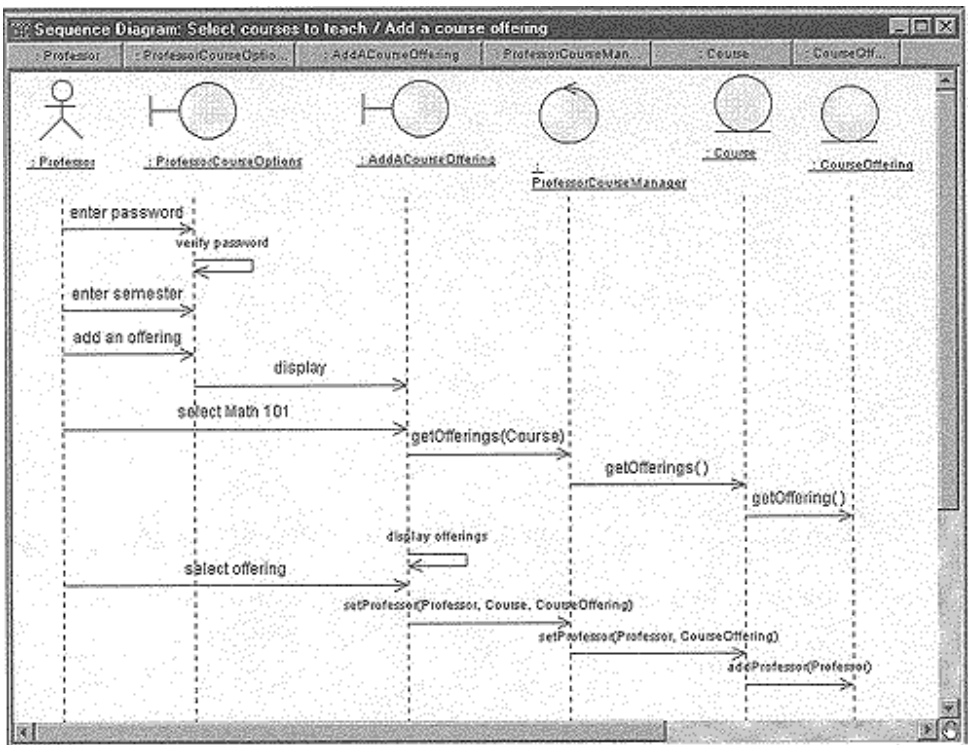


Figure 7-1 Sequence Diagram with Operations

Operations may also be created independently of interaction diagrams, since not all scenarios are represented in a diagram. This is also true for operations that are created to "help" another operation. For example, a course might have to determine if a specified professor is certified to teach it before it adds the professor as the teacher. An operation called *validate Professor ()* could be an operation created to perform this behavior.

Creating Operations in Rational Rose

1. Right-click to select the class in the browser and make the pop-up menu visible.
2. Select the New: Operation menu choice. This will create an operation called opname in the browser.
3. With the new operation selected, enter the desired name.

Operations for the Course class are shown in Figure 7-2.

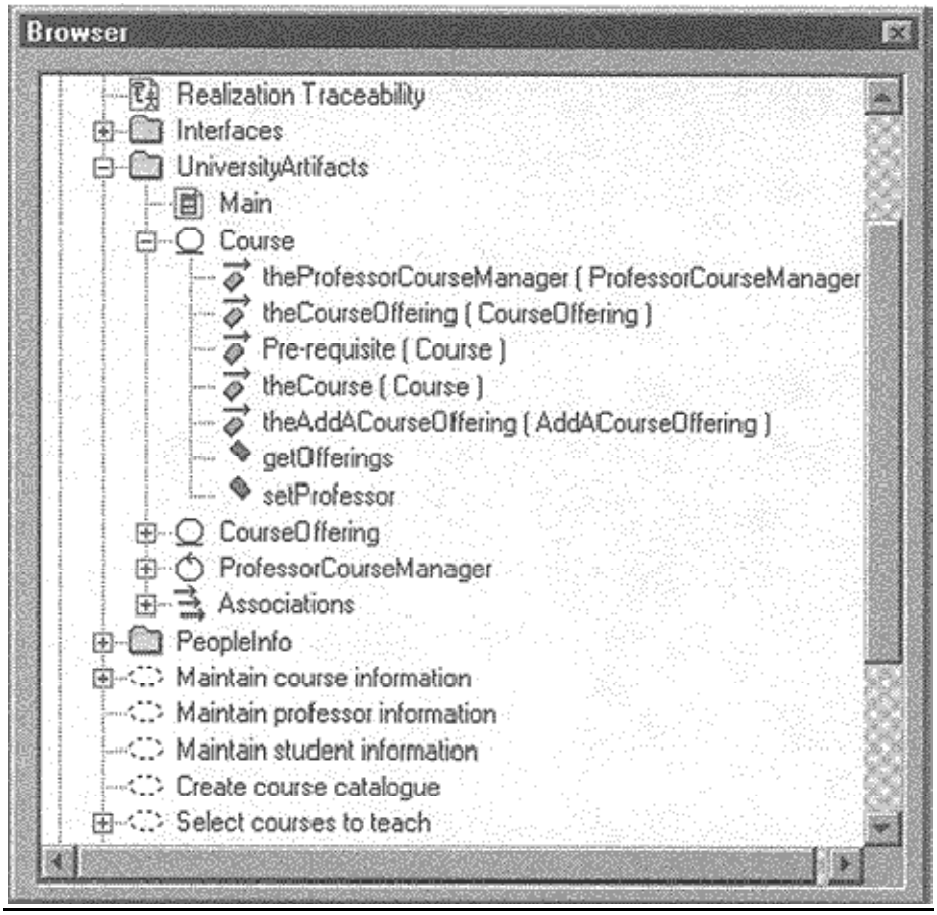


Figure 7-2 Operations in the Browser

Documenting Operations

Each operation should be documented so the reader of the model can quickly understand its purpose. The documentation should state the functionality performed by the operation. It should also state any input values needed by the operation along with the return value of the operation. The input and return values constitute the operation

signature. This information may not be known initially. Instead, it may be added later in the life cycle when more is known about the class.

Documenting Operations in Rational Rose

1. Click the + next to the class in the browser to expand the class.
2. Click to select the operation.
3. Position the cursor in the documentation window and enter the documentation for the operation.

The documentation for the *setProfessor()* operation of the Course class is shown in Figure 7-3.

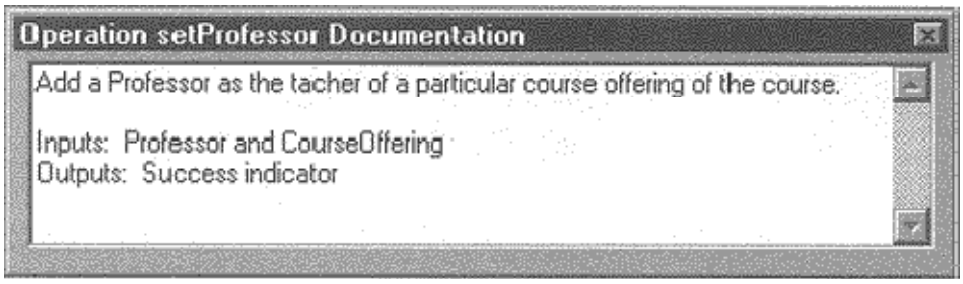

Operation setProfessor Documentation
Add a Professor as the tacher of a particular course offering of the course.
Inputs: Professor and CourseOffering
Outputs: Success indicator

Figure 7-3 Operation Documentation

Creating Attributes

Many of the attributes of a class are found in the problem statement, the set of system requirements, and the flow of events documentation. They may also be discovered when supplying the definition of a class. Finally, domain expertise is also a good source of attributes for a class. For example, the requirements state that information such as course name, description, and number of credit hours is available in the Course Catalog for a semester. This implies that name, description, and number of credit hours are attributes of the Course class.

Creating Attributes in Rational Rose

1. Right-click to select the class in the browser and make the pop-up menu visible.

2. Select the New: Attribute menu choice. This will create an attribute called Name in the browser.
3. With the new attribute selected, enter the desired name.

Attributes for the Course class are shown in Figure 7-4.

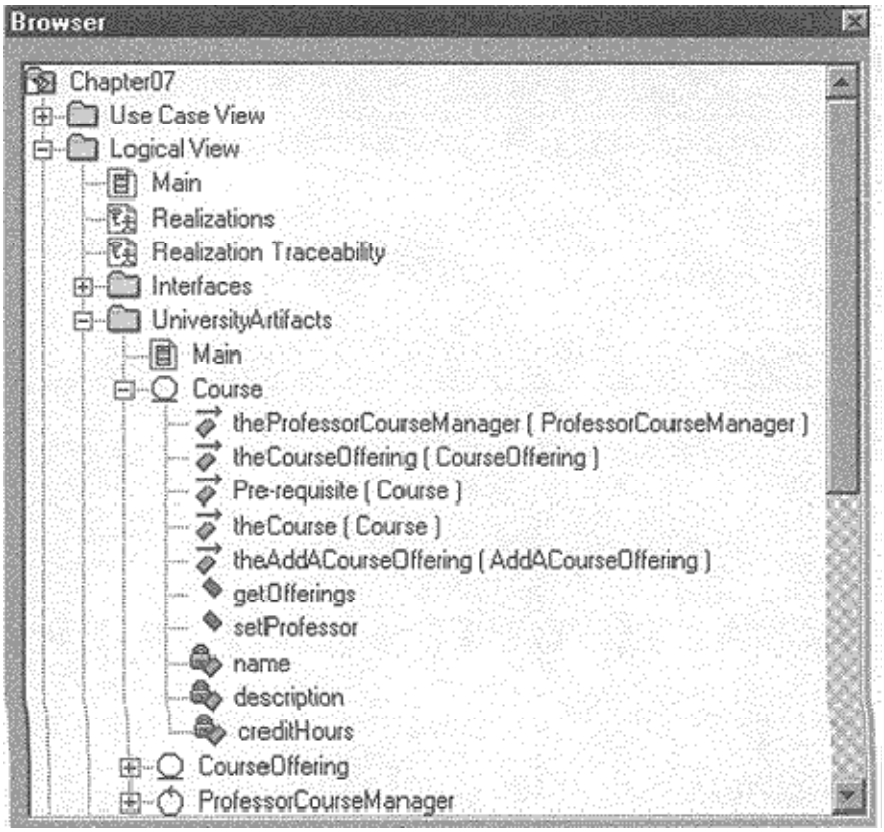


Figure 7-4 Attributes in the Browser

Documenting Attributes

Attributes should also be documented with a clear, concise definition. The definition should state the purpose of the attribute, not the structure of the attribute. For example, a bad definition for the name attribute of the Course class could be "a character string of length 15." A better definition for the name attribute is "The title of the course that is used in all University publications."

Documenting Operations in Rational Rose

1. Click the + next to the class in the browser to expand the class.
2. Click to select the attribute.
3. Position the cursor in the documentation window and enter the documentation for the attribute.

The documentation for the name attribute of the Course class is shown in Figure 7-5.

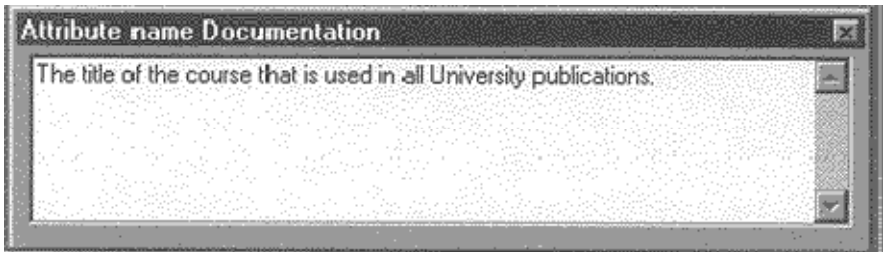


Figure 7-5 Attribute Documentation

Unit-4 Lecture 26

Displaying attributes and operations, Association Classes

Displaying Attributes and Operations

Attributes and operations may be displayed on a class diagram. Often, a class diagram is created specifically for this purpose—it shows the structure and behavior of the classes in a package. Relationships typically are not shown on this diagram.

Creating a Class Diagram to show the Attributes and operations for a package

1. Right-click to select the package in the browser and make the shortcut menu visible.
2. Select the New: Class Diagram menu choice. A class diagram called New Diagram will be added to the browser.
3. With the new diagram selected, enter the name of the diagram.

Adding Classes to a Diagram Using the Query Menu

1. Double-click on the diagram in the browser to open the diagram.
2. Select the Query: Add Classes menu choice.
3. Select the desired package.
4. Click to select the desired classes and click the >>>> button to add the classes to the diagram or click the All >> button to add all the classes to the diagram.

Filtering Relationships in Rational Rose

1. Double-click on the diagram in the browser to open the diagram.
2. Select the Query: Filter Relationships menu choice.
3. Click the None button in the Type field to hide all relationships shown on the open diagram.
4. Click the OK button to close the Relations window.

Displaying some Attributes or Operations in Rational Rose

1. Right-click to select the class on an open class diagram and make the shortcut menu visible.
2. Select the Options: Select Compartment Items menu choice.
3. Click to select the attributes and operations to be displayed.
4. Click the >>>> button.
5. Click the OK button to close the Edit Compartment window.

Showing all Attributes and Operations in Rational Rose

1. Right-click on the class in a diagram to make the shortcut menu visible.
2. Select the Options:Show All Attributes menu choice to display all the attributes for the class.
3. Repeat step 1 and select the Options:Show All Operations menu choice to display all the operations for the class.

NOTE:- To always display the attributes and operations for a class, you can set the Show All Attributes and Show All Operations selections using the Tools:Options menu.

Setting Stereotype Display in Rational Rose

1. Right-click on the class in a diagram to make the shortcut menu visible.
2. Select the desired Options:Stereotype Display menu choice (None = do not display stereotype, Label = show stereotype in << >>, Icon = show class using Stereotype icon).

The class diagram called Attributes and Operations for the University Artifacts package is shown in Figure 7-6. For this type of diagram I prefer to show the stereotypes of the classes as labels.

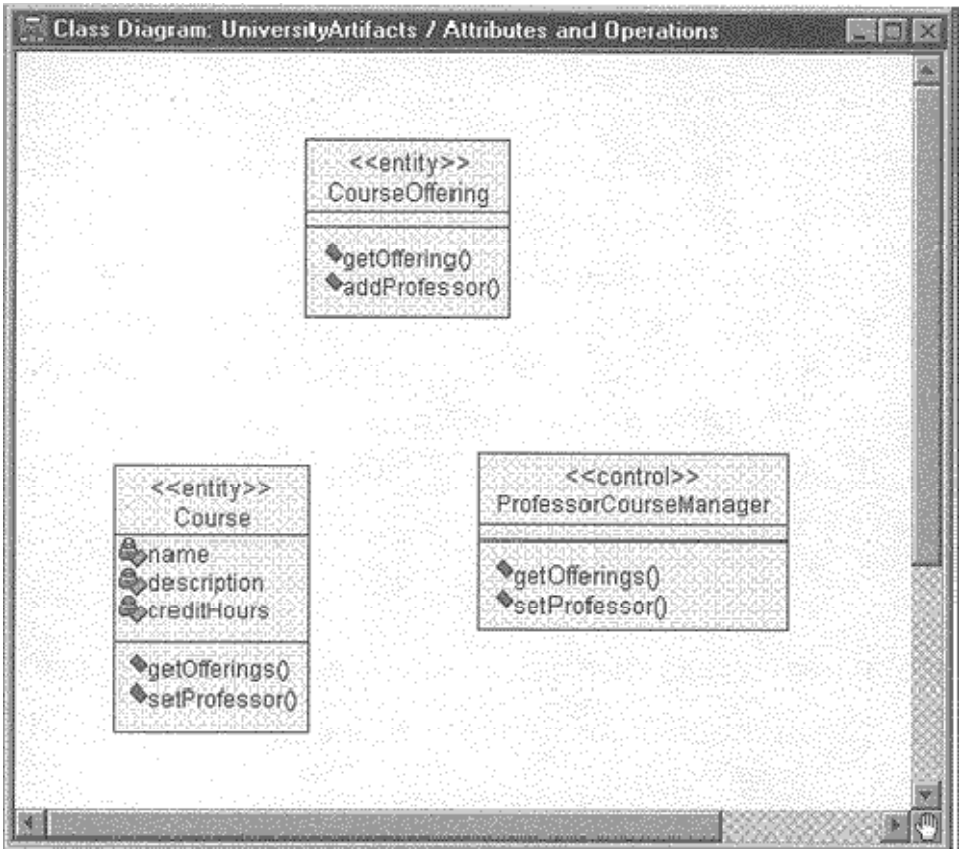


Figure 7-6 Displaying Attributes and Operations

Association Classes

A relationship may also have structure and behavior. This is true when the information deals with a link between two objects and not with one object by itself. Consider the following example: A student may take up to four course offerings, and course offering may have between three and ten students. Each student must receive a grade for the course offering. Where is the grade held? It doesn't belong to the student since a student will probably have different grades for different course offerings, and it doesn't belong to the course offering since a course offering has different grades for different students. The information belongs to the link between a student and a course offering. This is modeled as an association class. Since an association class behaves like any

other class, it too can have relationships. Following our example, a student receives a report card each semester that is made up of all the linked grade objects for the student.

Creating Association Classes in Rational Rose

1. Click to select the Class icon from the toolbar.
2. Click on the diagram to place the class.
3. With the class selected, enter the name of the class.
4. Add the attributes and operations to the class.
5. Click to select the Association Class icon from the toolbar.
6. Click on the association class and drag the association class line to the association it modifies.
7. Create any additional relationships for the association class.

The association class Grade is shown in Figure 7-7

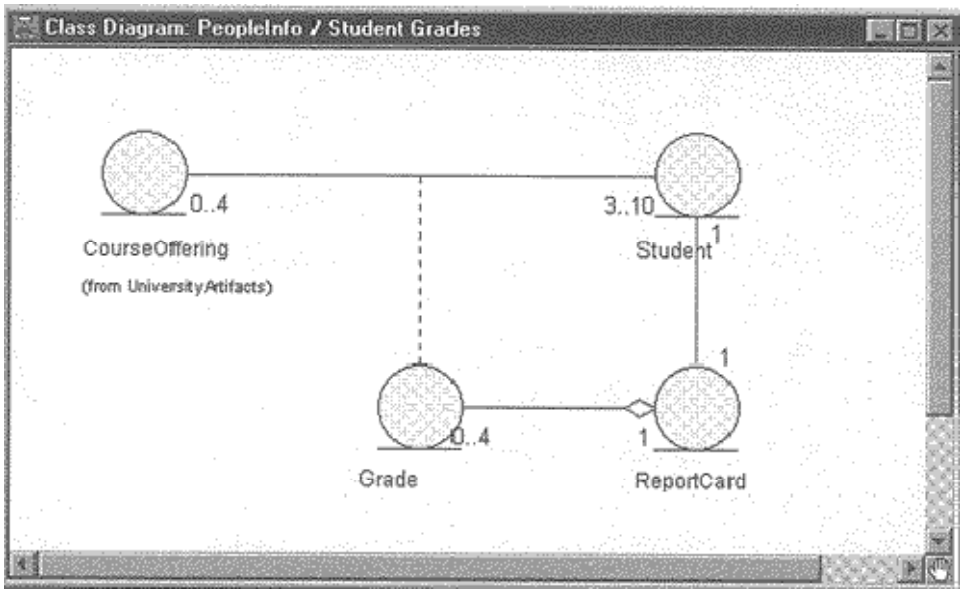


Figure 7-7 Association Class

Unit-4 Lecture 27

Analyzing Object Behavior

Modeling Dynamic Behavior

Use cases and scenarios provide a way to describe system behavior; that is, the interaction between objects in the system. Sometimes it is necessary to look at the behavior inside an object. A state chart diagram shows the states of a single object, the events or messages that cause a transition from one state to another, and the actions that result from a state change.

A state chart diagram will not be created for every class in the system, only for classes with "significant" dynamic behavior. Interaction diagrams can be studied to determine the dynamic objects in the system—ones receiving and sending many messages. State chart diagrams are also useful to investigate the behavior of an aggregate "whole" class and of control classes.

Care must be taken to stay in an analysis frame of mind—concentrating on the WHAT of the problem and not the HOW of the solution.

Creating state chart diagrams in Rational Rose

1. Right-click to select the class in the browser and make the shortcut menu visible.
2. Select the New: State chart Diagram menu choice. This will add a state diagram called New Diagram to the browser.
3. While the diagram is still selected, enter the name of the diagram.
4. To open the diagram, click the + to expand the class in the browser, click the + to expand the State/Activity Model in the browser and double-click on the state chart diagram in the browser.

The browser view of the state chart diagram for the Course Offering class is shown in Figure 9-1.

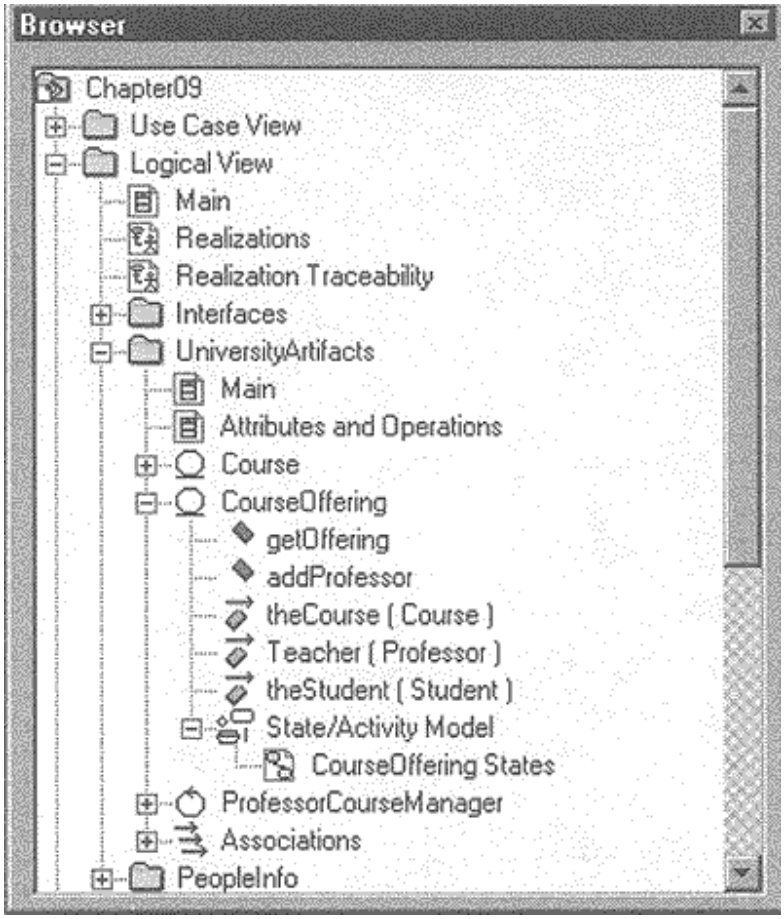


Figure 9-1 State chart Diagram in the Browser

States

A state is a condition during the life of an object during which it satisfies some condition, performs some action, or waits for an event. The state of an object may be characterized by the value of one or more of the attributes of the class. For example, a CourseOffering object may be open (able to add a student) or closed (maximum number of students already assigned to the CourseOffering object). The state depends upon the number of students assigned to the particular CourseOffering object. Additionally, a state of an object may be characterized by the existence of a link to another object. A professor may be teaching or on sabbatical. This depends upon the existence of a link to a CourseOffering object. Looking at the state of an object can validate the multiplicity chosen for a relationship to another object. That is, if being in a state depends upon the existence of a link to another object, this implies that the multiplicity of the relationship modifying the role of the associated class must include zero (i.e., the relationship is optional). Thus, the states of an object are found by examining the attributes and links defined for the object.

The UML notation for a state is a rectangle with rounded corners as shown in Figure 9-2.

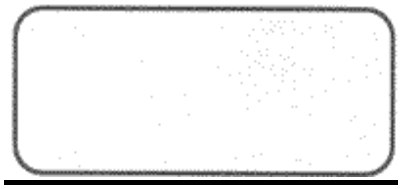


Figure 9-2 UML Notation for a State

A state chart diagram encompasses all the messages that an object can send and receive. Scenarios represent one path through a state chart diagram. The interval between two messages sent by an object typically represents a state. Therefore, sequence diagrams may be examined to discover the states for an object (look at the space between the lines representing messages received by the object). If we had created sequence diagrams for each use case in the ESU Course Registration system, we would discover that objects in Course Offering class can be in one of the following states: Initialization (created prior to registration but students have not been added to it), Open (able to accept students), Closed (maximum number of students already registered for it), Canceled (no longer offered).

Creating states in Rational Rose

1. Click to select the State icon from the toolbar.
2. Click to place the state on the state chart diagram.
3. With the state still selected, enter the name of the state.

The states of the CourseOffering class are shown in Figure 9-3.

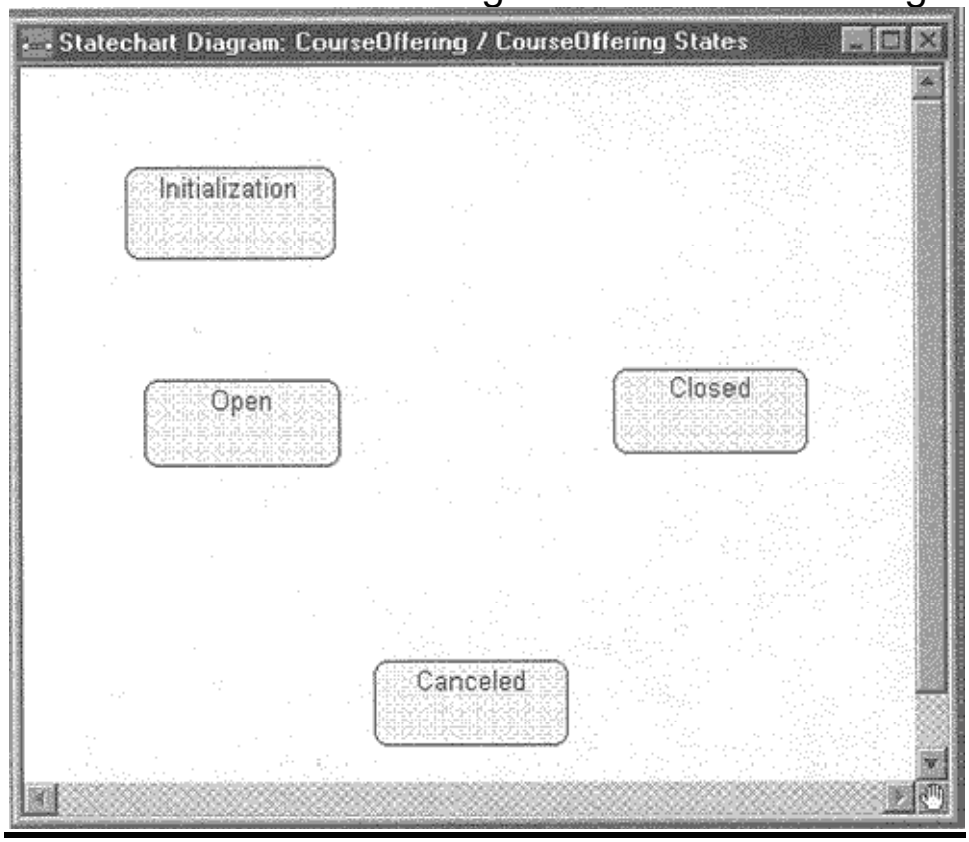


Figure 9-3 States

Unit-5 Lecture 28

Checking the Model

Making the Model Homogeneous

Why Homogenize

According to ***Webster's New Collegiate Dictionary****,* the word homogenizes means to blend into a smooth mixture, to make homogeneous. As more use cases and scenarios are developed it is necessary to make the model homogeneous. This is especially true if multiple teams are working on different parts of the model. Since use cases and scenarios deal with the written word, people may use different words to mean the same thing or they may interpret words differently. At this time classes may be combined, classes may be split into multiple classes, or a class may be eliminated from the model. This is just the natural maturation of the model during the project life cycle.

Homogenization does not happen at one point in the life cycle—it *must* be on-going. Projects that wait until the end to synch up information developed by multiple groups of people are doomed to failure. I have found that the most successful projects are made up of teams that have constant communication mechanisms employed. Communication may be as simple as a phone call or as formal as a scheduled meeting—it all depends on the project and the nature of the need to talk (for example, is it a simple question or a formal review of a piece of the system?). The only thing that matters is the fact that the teams are not working in isolation.

Combining Classes

If different teams are working on different scenarios, a class may be called by different names. The name conflicts must be resolved. This is accomplished mainly through model walk throughs. Examine each class along with its definition. Also examine the attributes and operations defined for the classes, and look for the use of synonyms. Once you determine that two classes are doing the same thing, choose the class with the name that is closest to the language used by the customers.

Pay careful attention to the control classes created for the system. Initially, one control class is allocated per use case. This might be overkill—control classes with similar behavior may be combined. Examine the sequencing logic in the control classes for the system. If it is very similar, the control classes may be combined into one control class.

In the ESU Course Registration System there is a control class for the *Maintain Course Information* use case and one for the *Create Course Catalog* use case. Each control class gets information from a boundary class and deals with course information. It is possible that these two control classes could be combined since they have similar behavior and access similar information.

Unit-5 Lecture 29

Splitting Classes

Classes should be examined to determine if they are following the golden rule of OO, which states that a class should do one thing and do it really well. They should be cohesive; for example, a Student Information class that contains information about the Student actor as well as information about what courses the student has successfully completed is doing too much. This is better modeled as two classes—Student Information and Transcript, with an association between them.

Often, what appears to be only an attribute ends up having structure and behavior unto itself and should be spilt off into its own class? For example, we'll look at Departments in the university. Each Course is sponsored by a Department. Initially, this information was modeled as an attribute of the Course class. Further analysis showed that it was necessary to capture the number of students taking classes in each department, the number of professors that teach department courses, and the number of courses offered by each department. Hence, a Department class was created. The initial

attribute of Department for a Course was replaced with an association between Course and Department.

Eliminating Classes

A class may be eliminated altogether from the model. This happens when:

1. The class does not have any structure or behavior

2. The class does not participate in any use cases

In particular, examine control classes. Lack of sequencing responsibility may lead to the deletion of the control class. This is especially true if the control class is only a pass through— that is, the control class receives information from a boundary class and immediately passes it to an entity class without the need for sequencing logic.

In the ESU Course Registration System, initially we would create a control class for the *Select Courses to Teach* use case. This use case provides the capability for professors to state what course offerings they will teach in a given semester. There is no sequencing logic needed for the control class—the professor enters the information on the GUI screen and the Professor is added to the selected offering. Here is a case where the control class for the use case could be eliminated.

Unit-5 Lecture 30

Consistency Checking

Consistency checking is needed since the static view of the system, as shown in class diagrams, and the dynamic view of the system, as shown in use case diagrams and interaction diagrams, is under development in parallel. Because both views are under development concurrently they must be cross-checked to ensure that different assumptions or decisions are not being made in different views. Consistency checking does not occur during a separate phase or a single step of the analysis process. It should be integrated throughout the life cycle of the system under development.

Consistency checking is best accomplished by forming a small team (five to six people at most) to do the work. The team should be composed of a cross-section of personnel— analysts and designers, customers or customer representatives, domain experts, and test personnel.

Scenario Walk-Through

A primary method of consistency checking is to walk through the high-risk scenarios as represented by a sequence or collaboration diagram. Since each message represents behavior of the receiving class, verify that each message is captured as an operation on the class diagram. Verify that two interacting objects have a pathway for communication via either an association or an aggregation. Especially check for reflexive relationships that may be needed since these relationships are easy to miss during analysis. Reflexive relationships are needed when multiple objects of the same class interact during a scenario.

For each class represented on the class diagram, make sure the class participates in at least one scenario. For each operation listed for a class, verify that either the operation is used in at least one scenario or it is needed for completeness. Finally, verify that each object included in a sequence or collaboration diagram belongs to a class on the class diagram.

Unit-5 Lecture 31

Event Tracing

For every message shown in a sequence or collaboration diagram, verify that an operation on the sending class is responsible for sending the event and an operation on the receiving class expects the event and handles it. Verify that there is an association or aggregation on the class diagram between the sending and receiving classes. Add the relationship to the

class diagram if it is missing. Finally, if a state transition diagram for the class exists, verify that the event is represented on the diagram for the receiving class. This is needed because the diagram shows all the events that a class may receive.

Documentation Review

Each class should be documented! Check for uniqueness of class names and review all definitions for completeness. Ensure that all attributes and operations have a complete definition. Finally, check that all standards, format specifications, and content rules established for the project have been followed.

Unit-5 Lecture 32

Designing the System Architecture

The Need For Architecture

Over the years I have heard many definitions of software architecture that range from "software architecture is what software architects do" to "software architecture is politics." I have come to the conclusion that software architecture is very difficult to define. It is a range of artifacts that are used to specify the strategic decisions about the structure and behavior of the system, the collaborations among the system elements, and the physical deployment of the system.

"Establishing a sound architectural foundation is absolutely essential to the success of an object-oriented project. Some teams try to ignore this phase, either because they are in such a rush to get a product out quickly they feel they don't have time to architect, or because they don't believe that architecting gives them any real value. Either way, the resulting head-long rush to code is always disastrous: fail to carry out this step properly and your project will likely experience software meltdown."

Architecture development is a very complicated issue. The architecture of the system is developed iteratively in the elaboration phase of development. "The architecture of a proposed system does not appear in a flash. It takes exploration of the use cases, a proof of- concept prototype, an architectural baseline, and other efforts during the Inception and Elaboration phases." Executable prototypes of the architecture are built to verify that the design decisions are correct. "Building something executable is absolutely essential, because it forces the development team to validate their design assumptions in the harsh light of reality."

The Architecture Team

Each project should have a chief architect who may be assisted by a small team of people. "The main activities of the architect include the definition of the architecture of the software, the maintenance of the architectural integrity of the software, the assessment of the technical risks of the project, the definition of the order and content of the successive iterations along with the planning of each iteration, providing consulting to various design, implementation, integration, and quality assurance teams and assisting in providing future market directions."

The 4 + 1 View Of Architecture

Software architecture is not a one-dimensional thing—it is made up of concurrent multiple views. Figure 11-1 shows the different views of software architecture. The rest of this chapter addresses the elements in each view of the architecture along with the UML notation used to represent the architectural decisions made for the system.

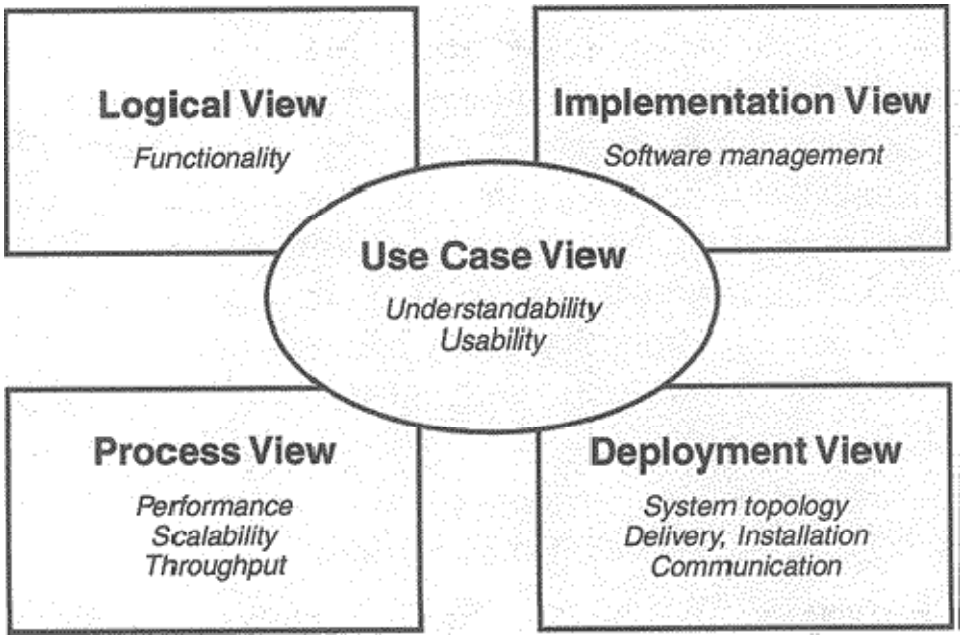


Figure 11-1 The 4+1 View of Architecture

Unit-5 Lecture 33

The Logical View

This view of architecture addresses the functional requirements of the system—what the system should provide in terms of services to its users. The logical architecture is captured in class diagrams that contain the classes and relationships that represent the key abstractions of the system under development. Most of the UML notation that has been addressed so far is contained within this view of architecture (e.g., classes, associations, aggregations, generalization, and packages).

This view of architecture is addressed early in the elaboration phase with the creation of classes and packages that represent the major abstractions of the domain. As time moves on, more classes and packages are added to the model to reflect the decisions made concerning the key mechanisms of the system. A key mechanism is a decision regarding common standards, polices, and practices. The selection of the key mechanisms for a system is often referred to as tactical design. "Poor tactical design can ruin even the most profound architecture, and so the team must mitigate this risk by explicitly identifying the project's key policies." Some common key mechanisms involve the selection of an implementation language, persistent data storage, the look and feel of the user interface, error handling, communication mechanisms, object distribution and object migration, and networking.

Today, many patterns exist that may be used to implement the key mechanism decisions made for your system. I strongly recommend looking into patterns before you try to "roll your own."

Additionally, the concepts of cohesion, closure, and reuse will affect the choices that you make. Robert Martin discusses some of the ramifications of the choice of packages for your system.

The bottom line is this: The UML may be used to communicate the strategic decisions made for your system by adding packages and classes to the model to communicate, implement, and document these decisions.

The Implementation View

This view of architecture concerns itself with the actual software module organization within the development environment. The implementation view of architecture takes into account derived requirements related to ease of development, software management, reuse, and constraints imposed by programming languages and development tools. The modeling elements in the component view of architecture are packages and components along with their connections.
A package in this view of architecture represents a physical partitioning of the system.
The packages are organized in a hierarchy of layers where each layer has a well-defined interface. The fact that an object-oriented system tends to be a layered system should not bring any surprises. This is due to the definition of an object—it should do one thing and do it well! A drawing showing some typical layers of a system may be found in Figure 11-3.

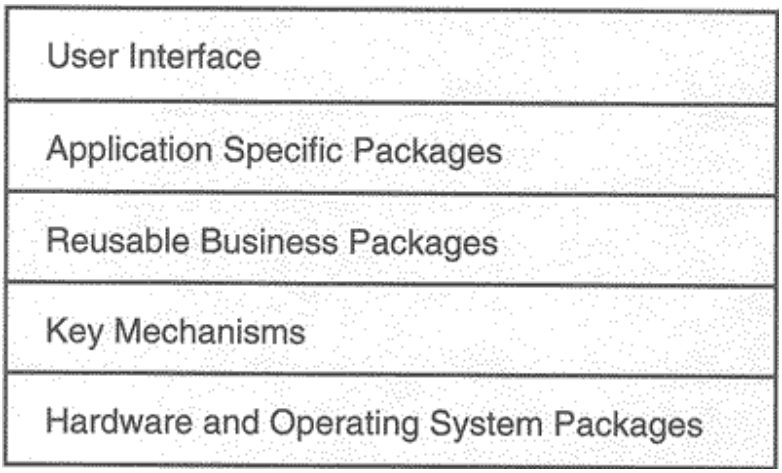


Figure 11-3 System Layers

The UML notation for a package in the component view is the same as a package in the logical view—a notched folder as found in Figure 11-4.

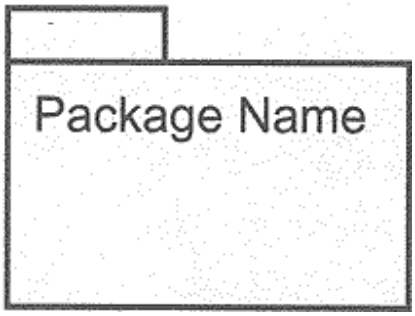


Figure 11-4 UML Notation for a Package

Creating Component View Packages in Rational Rose

1. Right-click to select the Component View package on the browser and make the shortcut menu visible.
2. Select the New:Package menu choice. This will add an item called NewPackage to the browser.
3. With the NewPackage still selected, enter the name of the package.

The main component diagram typically is a view of the packages defined for the system.

The Main Component Diagram in Rational Rose

1. Double-click on the Main Diagram under the Component View package on the browser to open the diagram.
2. Click to select a package and drag the package onto the diagram.
3. Repeat step 2 for each additional package.
4. Dependency relationships are added by selecting the Dependency icon from the toolbar, clicking on the package representing the client, and dragging the arrow to the package representing the supplier.

The main component diagram for the ESU Course Registration problem is shown in Figure 11-5

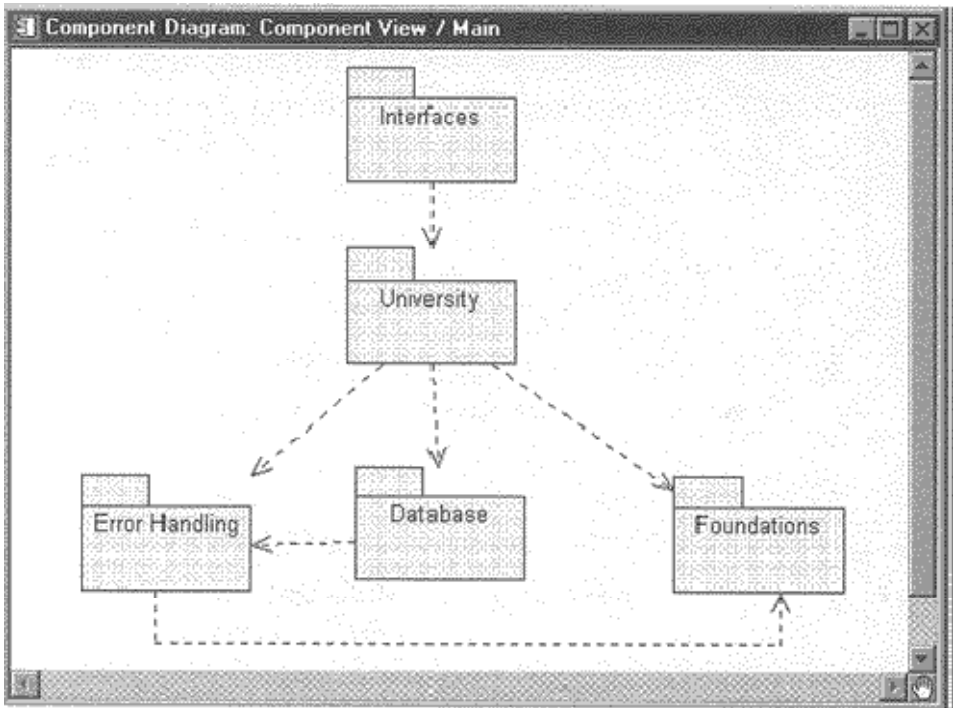


Figure 11-5 Main Component Diagram

Unit-5 Lecture 34

The Process View

This view of architecture focuses on the run-time implementation structure of the system. The process view of architecture takes into account requirements such as performance, reliability, scalability, integrity, system management, and synchronization. Components are also used in this view of architecture. Component diagrams are created to view the run-time and executable components created for the system. Components are related via dependency relationships. Run-time components show the mapping of classes to run-time libraries such as Java applets, Active-X components, and dynamic libraries. Executable components show the interfaces and calling dependencies among executables. Stereotypes may be used to visualize the type of

component. Interface classes that are created during design are shown using lollypop notation as shown in Figure 11-9.

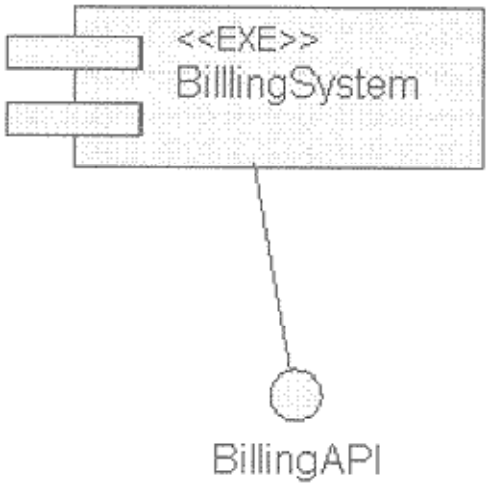


Figure 11-9 Component Interface

In Rational Rose, the information for the process view of architecture is created as diagrams in the Component View of the tool containing either run-time or executable components. Diagrams are created to show dependencies between the different types of components in the system.

For the ESU Course Registration System, the architecture team decided that there would be two DLLs—one containing course and course offering information and one containing database information. This allocation was chosen since it was felt that the course structure and the choice of database strategy was subject to change. By making them libraries, only the libraries would have to be replaced to implement future changes. There are three executables for the system—one for the Registrar to create and maintain the system, one for the Student to access the system, and one for the Professor to access the system. There is no communication between the executables. The component diagram for the Professor executable (ProfessorOptions.exe) is shown in Figure 11-10.

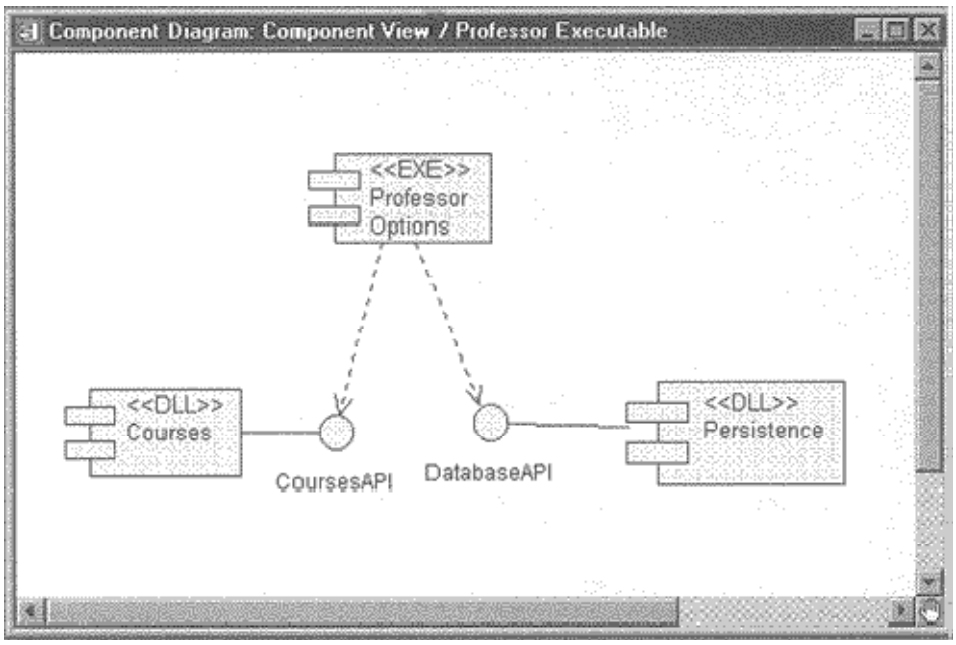


Figure 11-10 Professor Executable

The Deployment View

This view of architecture involves mapping software to processing nodes—it shows the configuration of run-time processing elements and the software processes living on them. The deployment view takes into account requirements such as system availability, reliability, performance, and scalability. Deployment diagrams are created to show the different nodes along with their connections in the system. The deployment diagram visualizes the distribution of components across the enterprise. Run-time processing elements are represented as nodes, which are connected by associations indicating communication paths between them. Software processes are illustrated as text attached to a node or group of nodes.

This diagram allows the architecture team to understand the system topology and aids in mapping components to executable processes. Issues such as processor architecture, speed, and capacity, along with inter-process communication bandwidth/capacity, physical location of the hardware, and distributed processing techniques, all come into play.

Deployment Diagram for The ESU Course Registration System

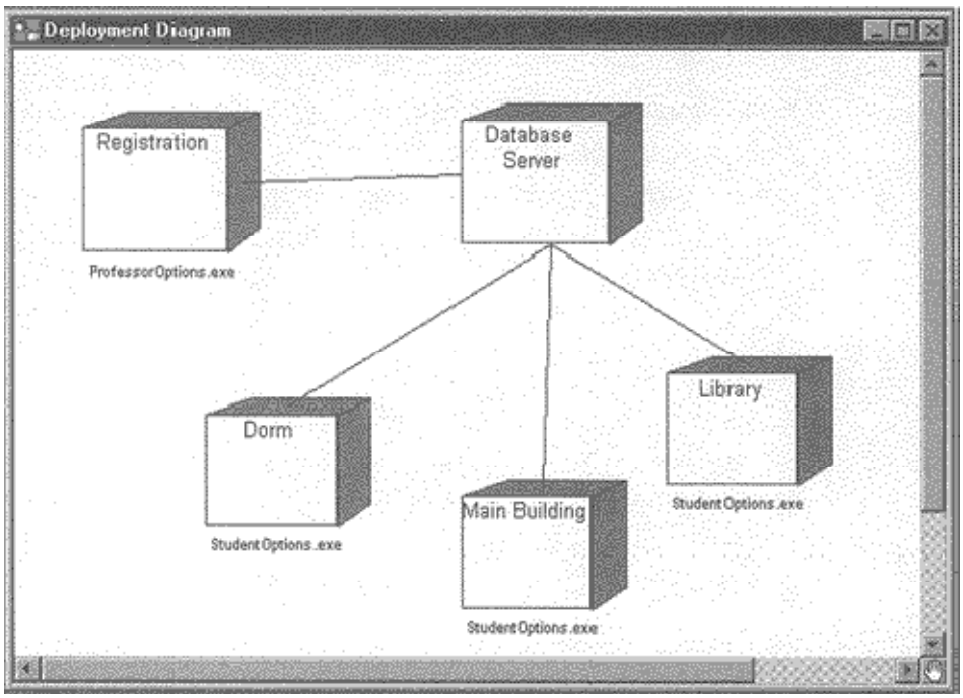
After studying the component packages defined for the problem, examining existing hardware, and estimating the load on the system during the course registration period, the architecture team decided that they will need five processors for the system—one to handle the professor executable, one for the database, and three for student registration.

Creating Deployment Diagram in Rational Rose

1. Rose automatically creates the deployment diagram. To open the diagram, double-click on the Deployment Diagram on the browser.
2. To create a node, click to select the Processor icon and click on the diagram to place the node.
3. With the node still selected, enter the name of the node.
4. To create a connection between nodes, click to select the Connection icon from the tool bar, click on one node on the deployment diagram, and drag the connection to the other node.

The deployment diagram for the ESU Course Registration problem is shown in Figure 11-11

Figure 11-11 Deployment Diagram



The Use Case View

This view of architecture demonstrates and validates the logical, process, component, and deployment views. Sequence diagrams and collaboration diagrams are created to show how the various design elements interact to produce the desired behavior.

Unit-6 Lecture 35

The Iteration Planning Process

The Iteration Planning Process

The iteration release plan prescribes schedules for all the increments of the system. "Such a plan must identify a controlled series of architectural releases, each growing in its functionality and ultimately encompassing the requirements of the complete production system."

"The iteration plan must state the iteration specific goals:

1. Capabilities being developed

2. Risks being mitigated during this iteration

3. Defects being fixed during the iteration

Exit criteria:

1. Updated capability information

2. Updated risk mitigation plan

3. A release description document, which captures the results of an iteration

4. Test cases and results of the tests conducted on the products including a list of defects

5. An iteration plan, detailing the next iteration including measurable evaluation criteria for assessing the results of the next iteration(s)"

The scenarios developed during analysis are the main input to this phase of development. The scenarios are examined and prioritized according to risk, importance to the customer, and the need to develop certain basic scenarios first. This task is best accomplished with a team made up of a domain expert, analysts, the architect, and testing personnel. "Scenarios should be grouped so that for each release, they collectively provide a meaningful chunk of the system's behavior and additionally require the development team to attack the project's next highest risks." As each iteration is completed, risks are reevaluated and the project plan is updated as needed. "For most projects, plan on about five (plus or minus two) intermediate releases."

The iteration planning process is shown in Figure 12-1.

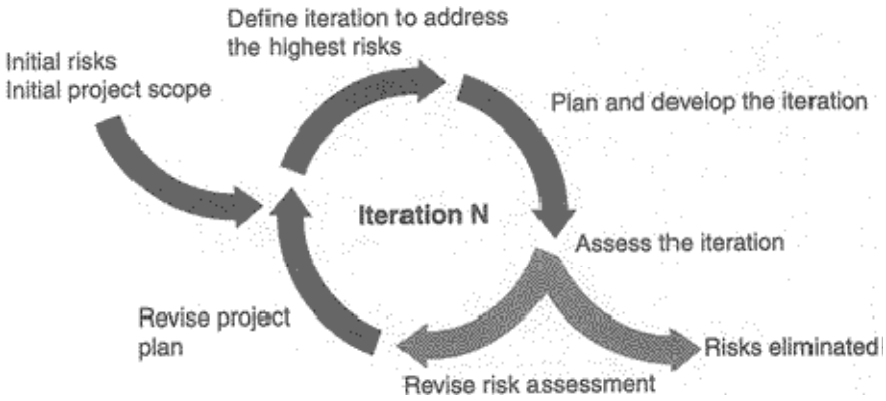


Figure 12-1 Iteration Planning Process

ESU Course Registration Problem Iteration Plan

For the ESU Course Registration problem the iteration plan is as follows:

Iteration 1

1. Maintain professor information
2. Select courses to teach
3. Maintain curriculum

Iteration 2

1. Maintain student information
2. Generate catalog

Iteration 3

1. Register for courses
2. Request class roster

Iteration 1 addresses the database risk—the courses must be stored in a database that is accessible to all. The *Maintain Professor Info* and *Select Courses to Teach* scenarios are in this iteration since they must be completed before the catalog can be generated. Iteration 2 adds the functionality needed to register a student (the student information must be in the database and the catalog must be available for the students). Iteration 3 completes the system.

Designing The User Interface

Earlier in the project life cycle, placeholder classes were created for the boundary classes for the system. Now, these classes must be finalized—number of windows, window layout, and handling events from users. Sequence diagrams are good sources for user interface requirements. Something in the user interface must provide the capability to receive all "messages" coming from an actor. Something in the user interface must provide the capability to send all "messages" going to an actor.

Select Courses to Teach User Interface Design

The scenarios dealing with the *Select Courses to Teach* use case were examined and the following requirements were found:

1. The professor must enter a password to enter the system. A window prompting the professor for a password was created.

2. If the password is valid, the ProfessorCourseOptions window is displayed. The ProfessorCourseOptions window contains a field for the semester and for the following buttons: ADD COURSE, DELETE COURSE, REVIEW SCHEDULE, and PRINT SCHEDULE.

3. Windows for the addition, deletion, and review options were created.

It is important to note that the design of the system cannot be based on only one scenario—the design matures as more scenarios are developed. I have chosen to use one scenario to illustrate the Rational Unified process and the UML notation that may be used to communicate the design decisions.

The ADD COURSE option deals with adding a professor to a course offering as the teacher. This scenario needs a window to allow the professor to input the needed information—I called it the Addition window. The Addition window contains the following selections:

1. Course name field

2. Course number field

3. Course offerings scroll box

4. OK button

5. Cancel button

6. Quit button

Once the professor enters the course name and number the system will retrieve and display the course offerings. Additionally, the professor will be able to select an offering. When the OK button is pushed, a message is sent to the course object.

The actual implementation of the GUI depends upon the class library chosen and is beyond the scope of this book. This section was included so you would understand the implications of object-oriented GUI design. Most GUI design is accomplished via a GUI builder. In this case, once the GUI classes are created, the reverse engineering capabilities of Rational Rose may be used to add the classes to the model.

Unit-6 Lecture 36

Adding Design Classes

Classes are typically added to the model to facilitate the "how" of a system. For example, the course registration system states that a professor must enter a password that must be validated. A class that knows how to validate passwords is added to the system so this requirement may be implemented. As classes are added to the system, they are displayed on diagrams as needed.

An updated class diagram for the ESU Course Registration problem is shown in Figure 12-2. (Note: At this point in the life cycle, I tend to show stereotypes as a label because I am typically adding a lot of detail to the model. This is just my preference, you do not have to do this. To set stereotype display as a label as the default, select the Tools:Options menu choice, Diagram tab, and label radio button in the Stereotype display field.)

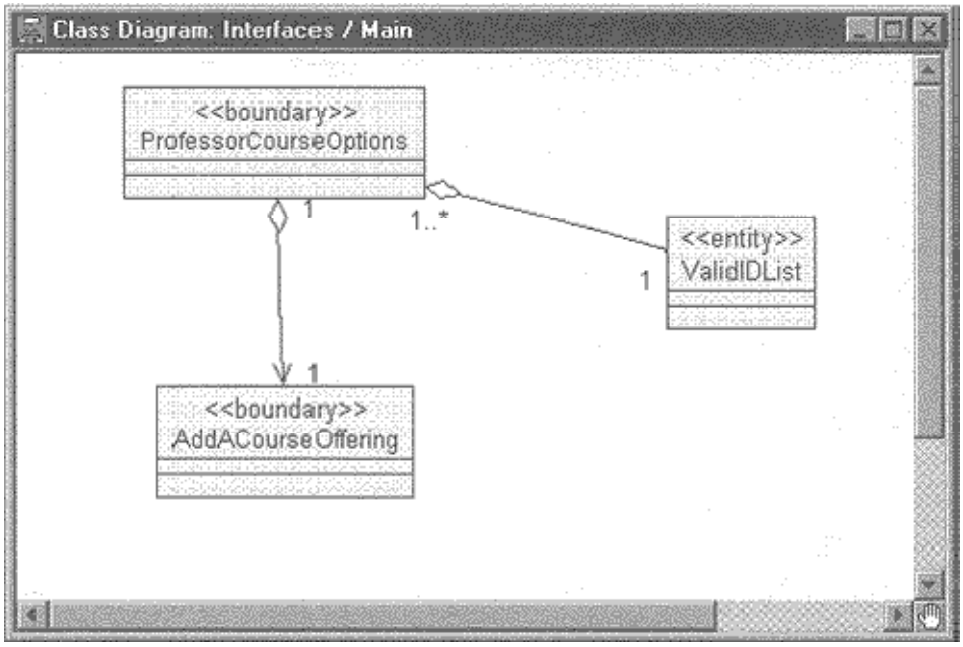


Figure 12-2 Design Level Class

The Emergence Of Patterns

Design patterns provide solutions to common software design problems. Thus, design patterns may play a part in designing the "how" of a system. In the words of Grady Booch, "patterns are, well, *very cool.*"[5] Patterns provide the capability to reuse successful designs and architectures, which leads to more maintainable systems and increased productivity. As with any classes developed at this point in the life cycle, the classes created to instantiate the design pattern are added to the model and to class diagrams. For example, the Abstract Factory pattern may be used to create the different types of RegistrationUser objects needed.

Unit-6 Lecture 36

Designing Relationships

The following design decisions must be made for relationships: navigation, containment,n refinement, and multiplicity implementation.

Navigation

Associations and aggregations are bidirectional relationships. During design, associations are revisited to determine if bi directionality is indeed needed. If possible, the relationship is made

unidirectional (i.e., navigable in only one direction) since unidirectional relationships are easier to implement and maintain.

Setting Navigation in Rational Rose

1. Right-click at the end of the association or aggregation line to be made non-navigable to make the shortcut menu visible.
2. Click to toggle the Navigation menu choice.

Several unidirectional associations are shown in Figure 12-3.

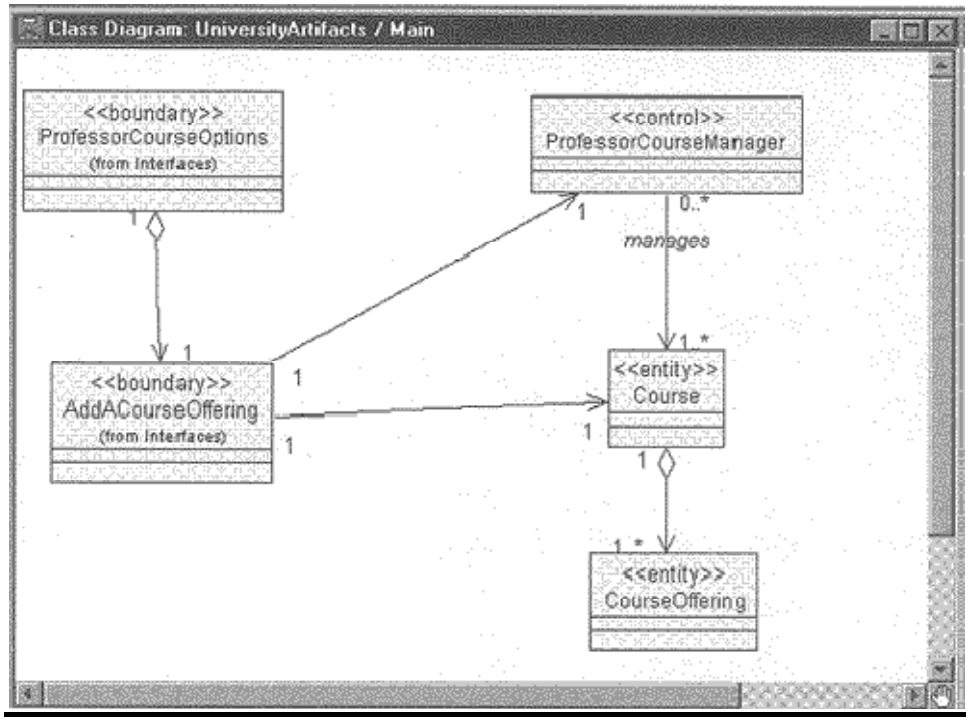


Figure 12-3 Navigation

Containment

Aggregation containment must also be added to the model. Containment may be by value or by reference. Containment by value implies exclusive ownership by the

containing class and is shown by a filled diamond. Containment by reference does not mandate exclusive ownership; it is shown by an open diamond.

Setting Aggregation Containment in Rational Rose

1. Double-click on the aggregation line to make the Specification visible.
2. Select the desired Detail tab for the role representing the aggregation whole.
3. Select the desired containment radio button.
4. Click the OK button to close the Specification.

A containment by value relationship (ProfessorCourseOptions contains AddACourseOffering) and a containment by reference relationship (ProfessorCourseOptions to ValidIDList) are shown in Figure 12-4.

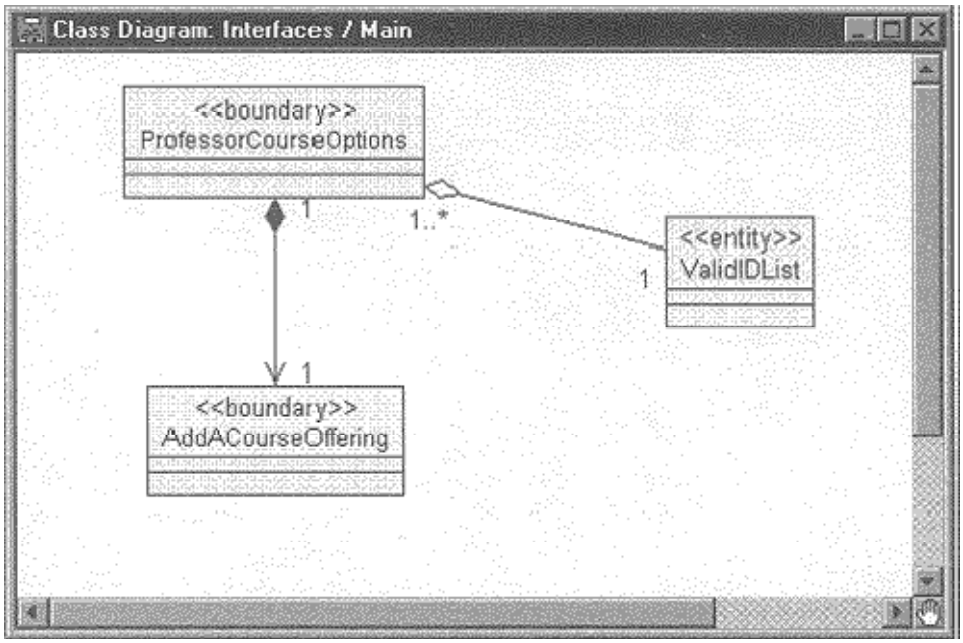


Figure 12-4 Containment

Refinement

An association relationship may be changed into a dependency relationship at this time. A dependency relationship implies that the object requesting the service (client) of another object (supplier) does not have intrinsic knowledge of the location of the supplier object—it must be told where the object is located. Typically, the supplier object is passed as a parameter to one of the methods of the client class or it is declared locally within a method of the client class. A dependency relationship is shown by a dashed arrow pointing from the client to the supplier.

Unit-6 Lecture 37

Designing Attributes And Operations

During analysis, it was sufficient to specify only attribute and operations names. During design, data types and initial values must be specified for attributes and operations signatures must be supplied. A data type for an attribute may be a language-supplied type such as an integer or it may be a more complex type like a String from a class library. If the attribute must be initialized to a specific value when an object of the class is created, this information is added to the class. In methodology terms, the operation signature includes the parameter list and the return class. Again, parameters and return classes must be assigned a data type. Access control—public, protected, and private must be set for attributes and operations. Attributes are typically private whereas operations may be private or public. If the class is part of an inheritance hierarchy, attributes and operations may be set to protected to allow access to subclasses.

Setting Attribute Data Types and Initial Values in Rational Rose

1. Right-click to select the class in the browser or on a class diagram and make the shortcut menu visible.
2. Select the Open Specification menu choice.
3. Select the Attributes tab.
4. Attributes will have a date type set to type and an initial value set to initval. Click to select the type or initval and place the field in edit mode.
5. Enter the desired date type or initial value.

Setting Operation Signatures in Rational Rose

1. Right-click to select the class in the browser or on a class diagram and make the shortcut menu visible.
2. Select the Open Specification menu choice.
3. Select the Operations tab.
4. Double-click on an operation to make the Operation Specification visible.
5. Enter the return class on the General tab.
6. Select the Detail tab.

7. Right-click in the Arguments field to make the shortcut menu visible.
8. Select the Insert menu choice. This will add an argument. Enter the name, data type, and default value for the argument.
9. Click the OK button to close the Operation Specification.
10. Click the OK button to close the Class Specification.
11. To display the operation signature on a class diagram, either set the display as the default using the Tools:Options menu choice, or the Diagram tab.
12. To display the operation signature on a class by class basis, select the class (es) and choose the Format: Show Operation Signature menu choice.

Note: Attribute and operation detail may also be set on a class diagram by selecting the displayed item and using the following format:

1. attribute : type = initial value

2. operation (argname : argtype = default) : return class

Some of the attribute and operation design decisions for the ESU Course Registration problem are shown in Figure 12-8.

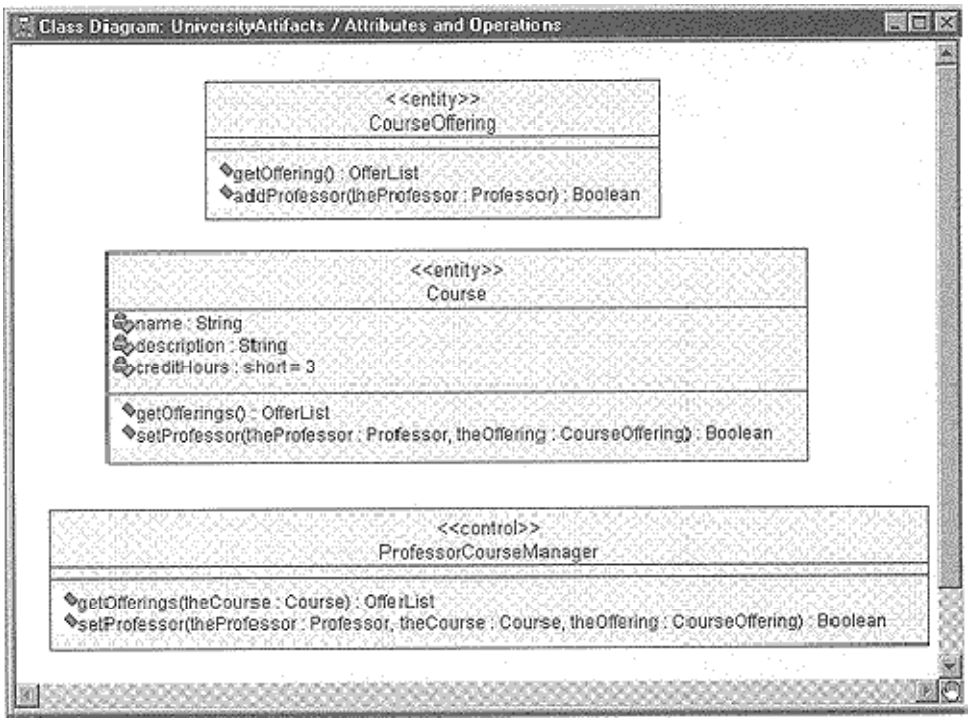


Figure 12-8 Attribute and Operation Design Details

Unit-6 Lecture 38

Designing For Inheritance

During analysis, inheritance hierarchies among key abstractions (i.e., classes) are established. During design, the inheritance hierarchies are refined to

1. Increase reuse

2. Incorporate design level classes

3. Incorporate chosen library classes

Analysis diagrams are reviewed to identify commonality of attributes, operations, and relationships. Super classes are defined to hold the newly discovered commonality. This reduces the amount of code to be written and tested. It also enforces uniformity; i.e., the same item cannot be handled differently in two different classes if the two classes inherit it from a common super class.

Analysis diagrams are reviewed to identify commonality of attributes, operations, and relationships. Super classes are defined to hold the newly discovered commonality. This reduces the amount of code to be written and tested. It also enforces uniformity; i.e., the same item cannot be handled differently in two different classes if the two classes inherit it from a common super class.

Code Generation And Design

The final step of design for an iteration is to add the methods that every good C++ class needs—for example, constructors, destructors, and copy constructors. This method can be added by hand, but this entails a lot of typing. This is why the ability to add these types of methods to a class is found in the Rose code generators.

Rational Rose has very powerful code-generation capabilities. Code is generated based on information obtained from the diagrams, the specifications, and the options specified in the code-generation properties for each type of element. Step-by-step guides to some of the code generation capabilities of Rational Rose may be found in the appendixes.

Unit-6 Lecture 39 and 40

Coding, Testing, And Documenting The Iteration

One of the final steps in building iteration is the implementation of the method bodies in the chosen language before iteration is complete. Interaction diagrams (Sequence and Collaboration diagrams) are used to help in this process because they tell you who does what to whom and when they do it.

Testing, although it is has not been discussed until this point, is a very important ingredient in the iterative and incremental life cycle. As the analysis and design progresses through the life cycle, testing plans and procedures are created. Again, use cases are an important part of this activity since they document what the system must do. The iteration should be tested to ensure that it truly does what is stated in the use case. Iterations are also integrated with previous iterations—you do not wait until the system is complete to put the iterations together. The iteration is evaluated to ensure that it eliminates its assigned risks. Any risks that were not eliminated (along with any risks that were found along the way) are reassigned to a future iteration.

The decisions made regarding the design of the iteration are captured in the models for the iteration. This information is used to generate the documentation for the iteration. Documentation should be generated on a iterative basis—I have found that systems

that wait until the project is complete to document it rarely have good documentation (indeed, they sometimes do not have any documentation!).

Using Reverse Engineering To Set The Stage For The Next Iteration

The model must be updated to reflect any changes made to the code (e.g., helping methods added, new classes added) while implementing the current iteration. Rather than updating the model by hand, the reverse engineering capability of Rose can be used to generate a model based on the current implementation, and this information can be merged into the model.